DOCUMENT RESUME

ED 084 833 EM 011 654

AUTHOR	Rubinoff, Morris
TITLE	Man-Machine Communication Through a Teletypewriter.
INSTITUTION	Pennsylvania Univ., Philadelphia. Moore School of Electrical Engineering.
SPONS AGENCY	Army Research Office, Durham, N.C.; National Science Foundation, Washington, D.C.
PUB DATE	May 73
NOTE	276p.
EDRS PRICE	MF-$0.65 HC-$9.87
DESCRIPTORS	Communications; Computer Graphics; Computers; *Indexing; Information Networks; Information Processing; *Information Retrieval; Information Storage; *Information Systems; Interaction; *Man Machine Systems; *On Line Systems; Program Descriptions; Programing Languages; Search Strategies; Telecommunication
IDENTIFIERS	Real English; SOLER Information System; *Teletypewriters

ABSTRACT

A ten-year research study designed a mechanized information system in the information processing field. Special attention was paid to implementation criteria entering into on-line retrieval through man-machine dialog from a remote typewriter or video terminal and four major areas were investigated: search strategies, machine stored indexer aids, disc file organization, and graphic displays. The final system developed, SOLER, is a powerful library-oriented information system permitting browsing through the data base, narrowing of the search to selected files, and further restricting to chosen segments of each file. SOLER is useful to experts, indexers, and searchers, using the Real English search language--a complete English grammar--to permit users to engage in a full dialog with the computer. Other major findings include the following: 1) there is a significant difference between information, which is broad and qualitative, and data, which is formal, specific, and quantitative; 2) there are significant differences between information handling in library versus problem-solving environments; and 3) computerized library systems can extend the scope of catalog information and provide helpful indexing tools to users. (Author/LB)

FILMED FROM BEST AVAILABLE COPY

ED 084833

University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
Philadelphia 19104

MAN-MACHINE COMMUNICATION THROUGH A TELETYPEWRITER

May 1973

The Moore School Information Systems Laboratory

Morris Rubinoff
Principal Investigator

The work described in this report has been supported by the U. S. Army Research Office-Durham and in part through the Air Force, the National Science Foundation, and in-house funds.

The Moore School Information
Systems Laboratory
University of Pennsylvania

EM 011 654

ED 084833

University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
Philadelphia 19104

MAN-MACHINE COMMUNICATION THROUGH A TELETYPEWRITER

May 1973

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

The Moore School Information Systems Laboratory

Morris Rubinoff
Principal Investigator

The work described in this report has been supported by the U. S. Army Research Office-Durham and in part through the Air Force, the National Science Foundation, and in-house funds.

The Moore School Information
Systems Laboratory
University of Pennsylvania

This is the best copy available of a document processed for EDRS by ERIC/EM. We are aware that some pages may not be readable in microfiche or hard copy. However, we feel that the document shouldn't be withheld on the basis of these pages alone.

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	1
	1.1 Summary of the Investigation	2
2.	THE SOLER SYSTEM	6
	2.1 General Description	6
	2.2 Basic Concepts	10
	2.2.1 Data Bases	11
	2.2.2 Files	12
	2.2.3 Records	14
	2.2.4 Interface Routines	15
	2.3 SOLER Commands	16
	2.4 Data Base Example	21
3.	THE INFORMATION SYSTEM COMPONENTS	26
	3.1 File Structure	27
	3.2 The Record Definition Phase	45
	3.3 Input Phase	57
	3.3.1 Operation of the Input Phase	57
	3.4 The Update Phase	63
	3.5 The Invert Phase	73
	3.6 The Retrieval Mechanism	81
	3.7 The Output Phase	106
4.	STATISTICAL CLUMPING AS AN INDEXING AID	114
	4.1 Adaptive Interaction	114
	4.2 Implementation of the System	118
	4.3 General Definitions	118

		Page
4.4	Theoretical Considerations	121
4.5	Preliminary Definition of an Adaptive System	125
4.6	Habit Forjing vs. Learning	126
4.7	A Working Definition of an Adaptive Process	130
4.8	The Final Form of the Adaptive Process	130
4.9	Interaction-by-Interaction Learning	132
4.10	The Interest Profile	134
4.11	An Algorithm for Rendering Adaptive Assistance	134
4.12	An Extension to the Adaptive Algorithm	137
4.13	The Implementation of the Interactive Process	138
4.14	Clumping - The Techniques Considered	149
4.15	Implementation	157
4.16	Results	162
5.	English as a Search Language	183
5.1	Easy English	183
5.2	Real English	187
6.	Bibliography	
6.1	Published Papers	205
6.2	Internal Reports	207
6.3	References on Information Retrieval	210
6.4	References on Statistical Clumping	214
6.5	References on Computerized English	215
6.6	References on Automatic Indexing	217
Appendix A		
	User's Manual	

MAN-MACHINE COMMUNICATION THROUGH A TELETYPEWRITER

This is the final report on a comprehensive research study of man-machine communication through a typewriter terminal. The study spans a full decade from 1962 through 1971 and has been supported in part through the U.S. Army Research Office - Durham and in part through the Air Force Office of Scientific Research, the National Science Foundation, and in-house funds.

The most significant findings of the study are as follows:

1. There is an essential difference between information and data. Information is by nature descriptive, qualitative, generic, and broad in coverage, and consequently hard to categorize crisply in single words or brief word phrases; data is formal, quantitative, specific and of narrow scope, and consequently lends itself to crisp tabulation or listing.

2(a). There is an essential difference between information handling in a library versus a problem-solving environment. A library is a repository of narrative descriptive information collected from a multitude of diverse independent sources; problem-solving uses of data collected by a single group of users for retrieval as part of the problem-solving process, i.e. on-line real-time retrieval.

2(b). As a consequence of 2(a), the computerized library information retrieval system need not be imbedded in a host system nor written re-entrant; the problem-solving data management system should be provided with both features.

3. The library information system does contain a conventional catalog card file but extended in scope in a computerized system to

accommodate author, title, sponsoring agency, publisher, date, etc., and bibliographical capabilities. The computerized library information system should also provide indexing tools, such as key words, notations of contents, index terms, etc., to help the searcher on the basis of the intellectual content of the files, concepts, techniques, procedures, and subject relationships.

4. The rate of growth of all literature is rapid. But the rate of growth in any one subject area is relatively slow. The value of a data base to each user decreases rapidly after his first search. The library information system is thus inherently a single-dip facility for individua_ , with the consequences stated in 2(b) above. However, the library information system is an effective information analysis and dissemination tool, to support the preparation of state-of-the-art summaries over broad subject areas or on specific topics tailored to the ever-changing needs of research. An on-line system of man-macine communication with a comprehensive data base, with appropriate tools for browsing and for sharpening the search, is an invaluable tools for such analysis and dissemination. The information system SOLER developed as part of this study is such a system.

1. SUMMARY OF THE INVESTIGATION

The investigation described here began in 1962 with the establishment of the Moore School Information Systems Laboratory to develop a design for a mechanized information system in the information processing field, with special attention to implementation criteria entering into on-line retrieval through man-machine dialog from a remote typewriter or video terminal. Four major areas were identified for study: search strategies; machine-stored indexes such as lists of index terms, synonyms, classification tables or "clumps", and other semantic aids; disk file organization; and graphic displays.

Progress was made in all four areas. The search strategy studies

led to deeper appreciation of man-machine communication problems and particularly to the difficulties encountered by the occasional searcher who does not have a computer background and has no reason or need to learn a specialized computer jargon in order to retrieve information from a data base. The Laboratory first designed, developed and tested a simple mechanization called "Easy English", whereby the user could gain access to the data base through readily recognized imperative sentences in natural English language. The remarkable success of Easy English led to the development of "Real English", whereby a complete English grammar was incorporated into the system and permitted the user to engage in a full dialog with the computer using declarative, imperative, interrogative, and even fragmented English sentences, restricted only by the limited vocabulary of the dictionary stored in the computer. Real English was completely successful as a search language; its only limitation was its demands on storage (close to 500,000 bytes). As computer memories become cheaper and faster, the use of Real English in information systems will become more economical and practical, and should find its place as a powerful tool in many information systems.

Machine-stored indexer aids have been studied by many investigators. Indexing is the most difficult aspect of the information retrieval process and probably the least understood. To begin with, in a private data library where the user collects his own files and indexes them himself for his own use, indexer aids are usually very primitive or totally unnecessary. Indexer aids are useful tools primarily in the realm of the public literature, where three different groups of people participate in the intellectual processes associated with the data base, namely, the experts, the indexers, and the searchers.

The experts are the authors, editors, and reviewers, who create the original information, establish the format and style in which the data is recorded, and determine which papers will appear in the body of knowledge and which will be rejected. The language of the literature is necessarily couched in the specialized terminology and concepts of the field as they exist at the time when the papers are written.

The indexers are library science specialists who are expected to understand and interpret the intellectual content of all the publications that get into the data base and to characterize each publication through (1) catalog identifications such as author, title, and publisher, (2) subject categorizers such as Dewey Decimal Number and subject heading(s), and (3) "index terms" such as "key words" in the title or in the text, and/or standardized words and word phrases that identify specific topics in the subject area, hopefully unambiguously. The indexers do their indexing a year or more after the papers have been written and the results are obviously influenced by the changes in outlook and terminology during the intervening period. In particular, the meanings that they intend to convey when they assign index terms often differ from the meanings of those terms at the times that the papers were written.

The searchers are of course those who have need for information which they hope can be found in the data base. These may be researchers who are entering a new field for the first time and seek information broadly over the field, or expert researchers who have worked in the field for some time and who are primarily interested in the most recent developments, or technical writers who are seeking state-of-the-art expositions at a semi-layman level, or reviewers who wish to check the state of the art before rating a new publication, or etc. Clearly, each searcher has

his own profile of interest, with different scope, depth, and level of technical sophistication, as well as relevant technical subject area. The searcher's problem is compounded by the staleness of the indexing since his search probably takes place years after the relevant publications were indexed.

Methodology for providing indexer aids was established in the course of this research investigation. The methodology is based on statistical generation of vocabulary relationships but it includes also the use of semantic tools, definitions, abstracts, and other narrative condensations of information.

Disk file organization has been through a number of revisions, each somewhat more sophisticated than the previous. At each stage, the file structure and organization were tested in a live experimental retrieval system. The final system, SOLER, is a particularly powerful library-oriented information system with the capability for browsing throughout the data base, narrowing the search to any selected files within the data base, and further restricting the search to selected on-line-identified segments of each file. The system permits the searcher to treat any word or word phrase anywhere in the data base as an index term and further permits him to search on terms related to his selected words or word phrases through classification tables previously derived by the computer from the information in the data base itself.

2. THE SOLER SYSTEM

2.1 General Description

The SOLER system is an information retrieval system. A user may store, modify, and retrieve information in either an interactive or background environment.

The basic unit of information handled by the system is called a data item. A social security number or a telephone number might be sample data items. A collection of logically related data items forms a record. For example, a record in a Personnel File will usually include data items for an employee's name, address, salary, position, etc. A file is simply a collection of records where the type of data items in each record are the same. The logical structure of the records is the same for each record in a file, but the data will vary. For example, if a set of records is collected to form a Personnel File, every record will contain a data item for the name of the employee; but the value or content of this data item will be different for each record. Thus the data item 'EMPLOYEE NAME' will occur in each record, and is part of the logical structure of the record; but the value of this data item will be different in each record. The following case might occur:

RECORD	DATA ITEM	VALUE
1	EMPLOYEE NAME	JOE JONES
2	EMPLOYEE NAME	MARY SMITH
3	EMPLOYEE NAME	JOE SMITH

It is often desirable to structure a record so that some pseudo-data items are defined which are then subdivided into several actual data items. For example, a record could be structured so that EMPLOYEE NAME is a pseudo-data item which contains two actual data items called 'FIRST NAME' and 'LAST NAME'. This structure is usually indicated by indentations as follows:

EMPLOYEE NAME	pseudo-data item
FIRST NAME	data item
LAST NAME	data item

This type of record structure is called a hierarchy or tree structure. In the SOLER system, pseudo-data items are called non-terminal data categories or nodes, and actual data items are called terminal categories or nodes. The previous example can now be re-written as:

RECORD	RECORD STRUCTURE	VALUE
1	EMPLOYEE NAME	---
	FIRST NAME	JOE
	LAST NAME	JONES
2	EMPLOYEE NAME	---
	FIRST NAME	MARY
	LAST NAME	SMITH
3	EMPLOYEE NAME	---
	FIRST NAME	JOE
	LAST NAME	SMITH

Referencing the non-terminal category EMPLOYEE NAME is interpreted as a reference to the two subordinate categories FIRST NAME and LAST NAME.

In the example above, some records have the same value for certain data items. Specifically, the FIRST NAME data item in both record 1 and record 3 has the value 'JOE'. In an information retrieval system, the user retrieves records by a request such as:

"Which employees have the first name 'JOE'?"

In this example, two records would satisfy the request. If the file contained many records, and it was necessary to check every record to determine if this condition was satisfied, the time needed for the search would be excessive. In the SOLER system, rapid retrieval times are possible because the system "inverts" the data file. That is, for a given value of a data item, the system will maintain a list of all the records in the file which contain that value in the specified category. For the sample data categories above, the lists would be:

CATEGORY	VALUE	LIST OF RECORDS
FIRST NAME	JOE	1,3
FIRST NAME	MARY	2
LAST NAME	JONES	1
LAST NAME	SMITH	2,3

When it is desired to retrieve a set of records containing certain data values, the inverted lists can be manipulated and an answer produced without referencing the actual data records until it is desired to display parts of the record. Since all needed information is in the inverted lists, the system can rapidly inform the user exactly how many

records satisfy the conditions specified in the request. When the records are actually retrieved, only those records which actually satisfy the request will be accessed. It is not necessary to examine any records which do not meet the given criterion.

EXAMPLE 1

To answer the question: "Which employees have the first name JOE?", the following commands should be presented to the SOLER system:

```
RETRIEVE FIRST NAME = JOE
PRINT EMPLOYEE NAME
```

By examining the inverted list for 'JOE', the system can immediately tell the user that 2 records satisfy the request. The PRINT command will then display the complete name for those employees. If the result of the RETRIEVE command is a large number of records, the user has the option of not printing any of the data, and trying another retrieval.

EXAMPLE 2

To answer the question: "How many employees are named JOE SMITH?", the following command would be used:

```
RETRIEVE FIRST NAME = JOE AND LAST NAME = SMITH
```

In this case, the answer would be produced through manipulation of the appropriate inverted lists. It is not necessary to reference the data records at all, unless the user decided to display data from the records identified by the search.

The SOLER system allows a user to specify the logical structure of his records directly from a terminal. He may then assign values to the terminal data items to enter as many records as desired into the file. When the file has been created, the user may perform searches of arbitrary complexity, and display portions of the resultant records directly on the terminal.

If the user encounters the common situation that he has several different files which contain logically related information, he may create a data base which contains as many as twenty-five files. These files may have different record structures, but it will be possible to retrieve simultaneously from any group of files in the data base.

The balance of this section describes the SOLER system in more detail. First the concepts of a data base, files, records, and data items are examined in more detail. Then the logical system modules are discussed, with an indication of the internal data flow in the system. The commands recognized by the system are outlined, and, finally, a detailed description of each system module is presented. Appendix A is a user's manual which describes how SOLER is used, what each command accomplishes, and illustrative examples which use every command in the SOLER system.

2.2 Basic Concepts

The SOLER system is designed to allow the user to create, update, and search a data base either from an interactive terminal or in a batch processing environment. The user may define as many data bases as desired. Each data base is self-contained and independent of all other data bases in the system. Within a data base, the user may

define as many as twenty-five separate files of information. These files may be logically connected to provide simultaneous searches spanning many different files of information. Records within a file have a flexible format, and may contain data in any format desired by the user.

2.2.1 Data Bases

A data base is the basic entity that may be manipulated by the SOLER system. The user may define as many data bases as desired, but only one may be accessed by the system at any one time. All data bases are completely independent and self-describing. That is, each data base contains within itself all the information necessary to allow the SOLER system to process the data base. Hence, once the user specifies which data base he would like to access, the system extracts any additional information needed for internal operations directly from the data base without requiring the user to re-specify any detailed information. Thus the user is able to initialize a data base only once, and from then on the system will automatically process the detailed internal information while the user directs the activities of the system at a logical level.

Each data base may contain anywhere from one to twenty-five files of information. For search operations, the user may manipulate any subset of the files as a logical group, and may define new logical groups, or delete old groups, at any time. Thus, the logical structure of the data base may be altered dynamically at the discretion of the user. This allows the user to view the data base as a single specific file, as a selected set of files, or as a single entity containing related data items.

2.2.2 Files

A file within a data base is a collection of logically related records. All records in a file have the same logical structure. As many as twenty-five files may exist within a single data base. The logical structure of each file is completely independent of all other files in the data base. The user may define several files with similar structures, or files with completely unrelated structures, depending on his own needs. All files within a single data base are logically interrelated through the system directories maintained by SOLER. The user may search across all files, or may specify that certain files be logically excluded from searches, thus searching a subset of the data base.

A file is defined by presenting to the system a description of the logical structure of the records to be contained in the file. This structure represents a tree of data categories. Any category which has one or more subordinate categories in the structure is referred to as a non-terminal node or category. A non-terminal category does not explicitly reference data, but implies a reference to any data contained in the nodes which are subordinate to the non-terminal node. A category which has no subordinate categories is a terminal category, and will explicitly contain data in a record.

In addition to the logical structure of the data, the user may assign a name of his own choosing to each data category. The name may be applied to more than one distinct category within the structure. In this case, use of the name will be automatically interpreted by the system as a reference to all data categories with that name. In addition to the

primary name of each category, the user may also designate as many synonyms for the name as desired. By specifying the name of one category as a synonym for another category, the user can define complex interrelationships between the categories within a file.

Data categories are said to be "linked" within the system under two conditions. All data categories subordinate to a given data category are termed "implicitly linked" because referencing the superior category implies a reference to each subordinate category. Data categories with the same name are termed "explicitly linked" since the user specified the same name for both categories. Also, all data categories subordinate to two explicitly linked categories are themselves implicitly linked.

Linking of categories is not restricted to a single file within the data base. If categories in two or more different files are assigned the same name (either as a primary name or a synonym), they also are explicitly linked. Thus the user may logically link data categories defined in separate files.

Note that categories which are explicitly linked through a given name may have other names which are not linked. Thus a reference by one name may imply two fields which are explicitly linked through that name. However, reference by another name may imply only one of the fields, or may link one of the original fields to another field. Thus the user may develop file definitions which have complex interlinkages both within each file itself and with other files in the data base. Conversely of course, the user may assign a unique name to every category so that no fields are explicitly linked.

During retrieval operations, referencing a category name normally references all explicitly and implicitly linked fields. Searches may thus simultaneously process multiple fields within one or more files. However, if the user desires to restrict the search to a more limited area of the data base, the name of a category may be qualified by appending one or more names of superior categories. This limits references to those categories in which all of the names appear as superior fields in the given order. In this way, the user may effectively unlink fields temporarily whenever desired. These qualifications may be introduced so that they remain in force until changed by the user, or a qualification may be explicitly stated to apply to a single command.

2.2.3 Records

A record in the SOLER system is a set of data items corresponding to the logical structure defined for one of the files in the data base. A record is a single occurrence of the logical pattern defined to the system when a file is created. As many records as desired may appear within a file, the only restriction being that the aggregate of all records in the data base may not exceed 16 million records.

Within each record, the data items correspond to the terminal categories of the file definition tree. The contents of each data item may be as large as 65,000 characters containing any data which can be processed by the machine (in general, this corresponds to the character sets specified by the ASCII or EBCDIC codes). Data items may occur more than once in a record (if the user has specified that a category may repeat), or may be omitted from a particular record. When data

items are omitted in a record, no space is allocated for the missing item, thus conserving space on the storage device on which the data base resides. At the present time, one record may contain a maximum of 1000 data items, but this limit may easily be expanded if the need arises.

The data stored within a data item may occur in any format. The system supplies routines to handle the standard fields of numeric data, character strings, and narrative text. If the user has a requirement for specialized processing of a particular data field, he may supply his own functional routine to the system. SOLER will invoke this routine whenever special processing of the data item is required. With this technique the user with one or more specialized requirements for the handling of data items may allow the standard system routines to handle the bulk of the processing, and all of the details of internal storage allocation, command interpretation, etc., while processing a few items in a manner prescribed by the user. A simple example would involve specifying a processor that would not display the salary data item in a personnel record until certain security checks have been validated.

2.2.4 Interface Routines

An important concept in the SOLER system is the idea of an interface routine.

It was desired to allow the system to handle any type of data that might arise in almost any application. Most existing information retrieval and data management systems allow flexibility in the structure of a record, but only limited data item formats are allowed. In most cases, the system will handle numbers and simple text strings, usually of limited length. The SOLER system, on the other hand, was designed

to allow any format of a data item desired by the user. The technique developed to accomplish this is the concept of an Interface Routine.

Basically, the SOLER system provides a superstructure for an information retrieval system. The system handles space allocation in the files, accesses to the I/O devices, detection and interpretation of commands, retrieval of records, etc. But at the same time, whenever it is necessary to process the internal format of a data item, an interface routine is called. For example, during execution of the Input Phase, the interface routines are responsible for reading the data from the external device and placing it in a buffer provided by the system. The system must know the length of the data; the format is immaterial. With this method, the SOLER system itself is independent of the format of the data stored in the system. The user may define a record structure which specifies the existence and logical relations of the data items, and may also supply interface routines to handle data items internally.

2.3 SOLER Commands

The full set of SOLER commands is presented here. A brief description of the action of each command indicates the purpose of the command and its uses and effects. A more complete description of system behavior is provided in Section 3 and in Appendix A.

Input Command

INPUT X (where X = file name)

initiates a question and answer dialogue with the computer in which the computer calls for all the data in a new record, one data field at a time, by field name; the user types the data item appropriate to that field and delivers it to the computer. The data fields are, of course, those which are permanently associated with file X.

Invert Command

INVERT X (where X = file name)

causes the computer to invert every record in file X that has not yet been inverted. The process of inverting a record is the addition of an entry into a list for each term (word, number, or series of alphanumeric characters) which appears anywhere in the record. Thus, the list for each term indicates every record in which that term appears.

Update Commands

DELETE X (where X = category name)

instructs the computer to delete all data in category X from the records in the active list (see RETRIEVE command).

MODIFY X (where X = category name)

instructs the computer to display to the user all data in category X (from only the records in the active list); the user modifies the data items displayed to him and returns them to the computer which automatically inputs and inverts them.

ADD X (where X = category name)

causes the computer to request new data appropriate to category X to be added to the records in the active list, record by record.

Field and File Locating Command

WHERE X (where X = term)

displays to the user the names of all fields (in the entire data base) in which term X appears as data, and the names of all files in which term X is the name of a category. The entire data base may be limited by the QUALIFY command.

Logical Retrieval Commands

RETRIEVE X (where X = logical expression of category names and terms)

produces a list (called the active list) of all the records which contain data that satisfies the conditions of logical expression X. There is only one active list; hence, every RETRIEVE command creates a list which replaces the previously active list. The logical expression X contains elements of the form:

category name = term.

The retrieved records may be limited by the RESTRICT command. The logical expression may be limited by the QUALIFY command.

APPLY U X (where X = logical expression of category names and terms, and
U = logical operator)

causes the computer to process a RETRIEVE command based on the conditions of logical expression X; then logical operator U is applied to this list and to the previous active list, thus producing a new active list. For example, if U is the operator "AND", then the new active list consists of all records which appeared on the previous active list and also appeared on the list produced by the processed command.

REPEAT

instructs the computer to reprocess the most recent APPLY or RETRIEVE command (after intervening commands have been implemented).

Search Limiting Commands

RESTRICT

instructs the computer to "remember" the records in the current active list; the searching done in all subsequent logical retrievals will be limited to those records which have been "remembered".

QUALIFY X (where X = set of file names and/or category names)

instructs the computer to "remember" the file names and category names; the searching done in all subsequent logical retrievals will be limited to those files and categories which have been "remembered". All subsequent output commands will be similarly limited.

Output Commands

PRINT N,X (where N = an integer and X = category name)

instructs the computer to print all data in category X from the next N records in the active list. The output is directed to the user's terminal. A pointer to an entry in the active list is maintained; this pointer determines the place in the list to begin printing the next N records. Whenever a new active list is created, the pointer is set to the beginning of the list; the pointer can be changed by using the FORWARD, BACKWARD, and RESET commands.

LIST N,X (where N = an integer and X = category name)

initiates the same processing as a PRINT command, except that the output is directed to the high-speed printer instead of the user's terminal.

CONTINUE N (where N = an integer)

instructs the computer to print all data in the category that was specified in the most recent PRINT or LIST command, from the next N records in the active list.

Active List Manipulating Commands

FORWARD N (where N = an integer)

moves the active list pointer forward N entries (or to the last entry on the list if N overshoots).

BACKWARD N (where N = an integer)

moves the active list pointer backward N entries (or to the beginning of the list if N overshoots).

RESET

sets the active list pointer to the beginning of the list.

SAVE X (where X = any name)

stores the current active list internally, under the identification name X. In order to reference it, a RESTORE X command must be used.

RESTORE X (where X = name used in any previous SAVE command)

replaces the current active list with the list identified by name X.

ERASE X (where X = name used in any previous SAVE command)

releases the list identified by name X; further reference to this list is no longer possible without re-retrieval.

GET X (where X = set of record numbers)

creates an active list composed of all records specified in set X.

Miscellaneous Commands

SET X (where X = collection of condition-setting pairs)

changes the settings of the conditions specified by collection X. The conditions pertain to the format of the user's dialogue with the computer. For example, the user can specify the output line length; the user can request to have all dialogue captured and permanently stored.

COMMENT X (where X = any comments)

stores comment X for the SOLER administrator to read at a later time.

END

ends any session on the SOLER system.

2.4 Data Base Example

A data base is defined in this section as an example of the storage and retrieval capabilities of the SOLER system. Since this is intended as an introductory example, no attempt is made to illustrate all capabilities of the system. The intent of the example is to indicate the range of logical file structures allowed, and the power inherent in the ability to link files. These examples will also be referenced in the more detailed discussions which follow, when the logical data flow within the SOLER system is investigated.

In this example, a data base is assumed called the Personnel Information Data Base. There are three separate files in this data base:

A Personnel file,

A Job Description file, and

A Definition file

The definitions for these files are shown below, where the tree structure is indicated by the indentation of the category names.

PERSONNEL FILE
- EMPLOYEE DATA
  - EMPLOYEE NAME
    - LAST NAME
    - FIRST NAME
    - MIDDLE INITIAL
  - ADDRESS
    - STREET
    - CITY
    - STATE
    - ZIP CODE
  - PHONE NUMBER, REPEAT
- JOB INFORMATION, REPEAT
  - JOB TITLE
  - LENGTH OF SERVICE
  - SALARY
  - OFFICE ADDRESS
  - PHONE EXTENSION
- EDUCATION, REPEAT
  - DEGREE
  - INSTITUTION ATTENDED
  - DATA RECEIVED
- AREA OF SPECIALIZATION, REPEAT
  - AREA
  - QUALIFICATIONS
- TECHNICAL PUBLICATIONS
  - TITLE
  - ABSTRACT

JOB DESCRIPTION FILE
- JOB TITLE
- QUALIFICATIONS
  - EDUCATION
  - EXPERIENCE
- SALARY
  - LOWEST LEVEL
  - HIGHEST LEVEL
  - AVERAGE

DEFINITION FILE
- WORD
- DEFINITION

The tree structure of this data base is shown in Figure 2.4.1. The word REPEAT after some of the data categories indicates to the system that the category will occur more than one time in many records. When a repeat is associated with a non-terminal category, the entire substructure of the record below this category will be repeated. It is also possible to "nest" repeats so that within a repeating structure, subordinate categories will also repeat. In Figure 2.4.1, repeating nodes are indicated by a dashed line leading to a second occurrence of the node. In an actual record, the field could repeat as many times as desired by the user.

In this data base, our hypothetical user is storing personnel information, a series of job descriptions, and also definitions of words. Within one data base, he has collected three interrelated files of information. When it is necessary to find a person for a certain position, the user can look at the qualifications of the job description, and then list the names of all employees who satisfy the requirements. It is also possible to search the data base for a specific job title. Since the JOB TITLE categories under the Personnel File and the Job Description File are explicitly linked, the user can display a description of the job, and also a list of all employees currently holding the position.

If a word in a job description or a technical paper is unfamiliar to the user, he can immediately refer to the dictionary for a definition.

This indicates some of the power provided to the user through the ability to link together files of diverse information within a single data base. These examples also indicate the variety of data types which

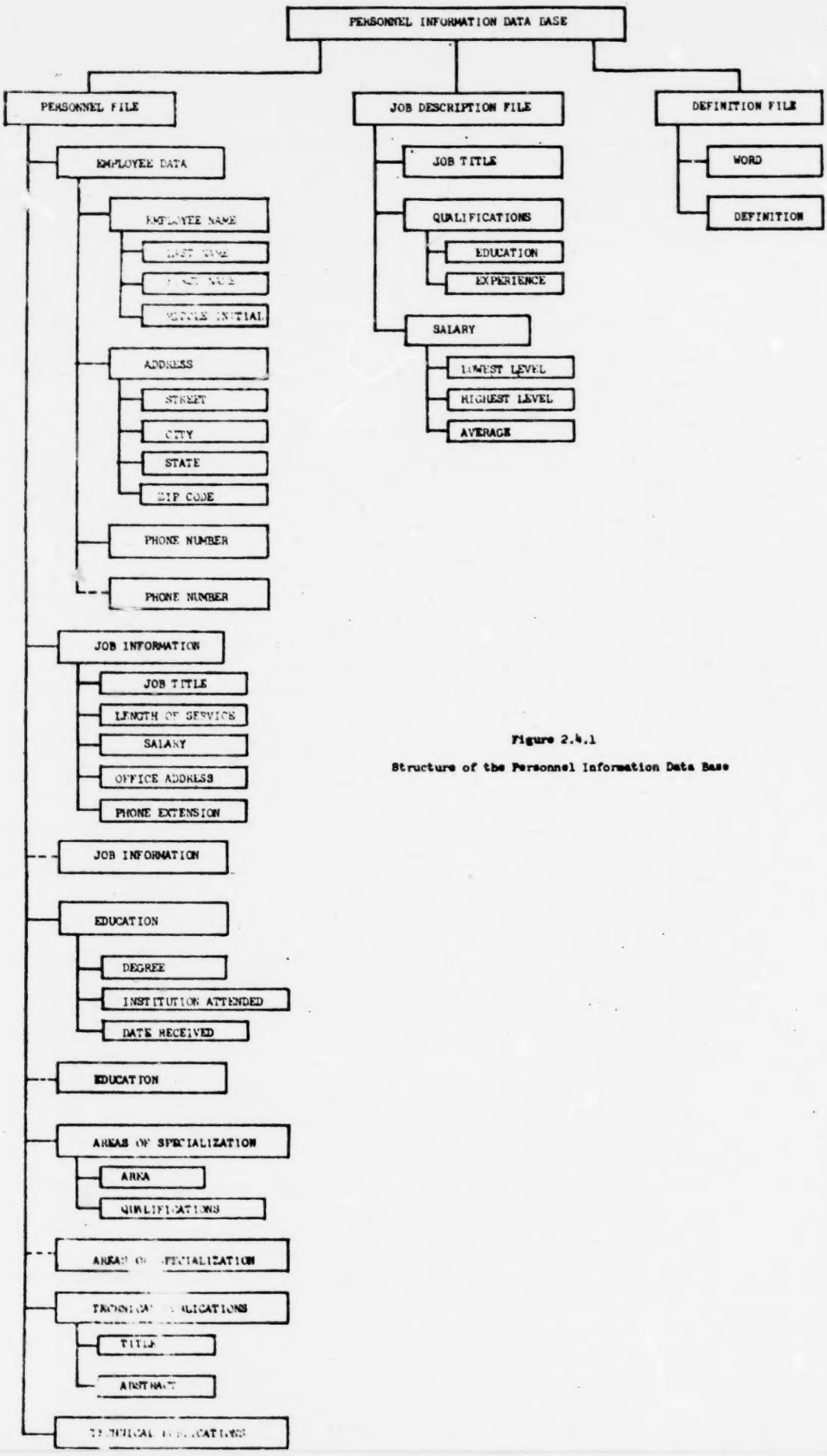


Figure 2.4.1

Structure of the Personnel Information Data Base

can be processed by the SOLER system. The Personnel File contains several well-defined fields, such as Salary and Phone Number, and also contains relatively unstructured data as free-form narrative text (in the abstract and title of publications).

3. THE INFORMATION SYSTEM COMPONENTS

The System for On-Line Entry and Retrieval (SOLER) has been implemented as a sequence of six basic phases; record-definition, input, update, invert, retrieve, and output. All of these phases can be executed interactively from a Teletypewriter, Datel/Selectric typewriter, or a video console (CRT or VDT). The unusually large variety of command options makes SOLER a highly versatile storage and search tool for natural-language text, bibliographic information, and all other types of library data. Written in COBOL, SOLER is readily transferrable to other computer systems; however, since the programs are not embedded in the operating system, they are less effective in a problem-solving environment.

The record definition phase (Section 3.2) permits the user to define and enter the logical structure of a new file. The input phase (Section 3.3) then permits him to enter data records into any file in the data base, including any newly created files. The update phase (Section 3.4) permits the user to add, delete, or correct data in the data base, either interactively or batch. The invert phase (Section 3.5) is a processor which forms inverted lists from the input records. The inverted lists serve to speed up the retrieval of records in the retrieve phase of the system (Section 3.6). The output phase (Section 3.7) provides for display of retrieved data on typewriter, high-speed printer, and/or video display console.

Section 3.1 presents a detailed description of the various files that are incorporated into SOLER. The particular choice of files was dictated by the requirement to provide a reasonable balance between user service and system efficiency.

3.1 File Structure

All file access is through one subroutine -- READWRIT, thus allowing machine independence. Files are classified into two general categories -- directory files and data files. The former are characterized by 3600 byte tracks and the latter by 2000 byte tracks -- the physical track size is the actual distinction between the two types. The track sizes are not flexible -- the two track lengths are built into the system at every turn and could not be changed within the existing implementation.

The system uses a set of five files. This is the minimum number needed to support the system, namely, the system file, a directory file, and one inverted, one direct data, and one work file. These are described below. Files may be added at any time -- this involves execution of several programs (see Appendix A) for physical and logical creation and storage of information describing the files. READWRIT must also be recompiled with FILE and DEFIN macros describing the new file. These macros create the File Control Block which will describe the file to TSOS during I/O operations, and internal tables for the subroutine.

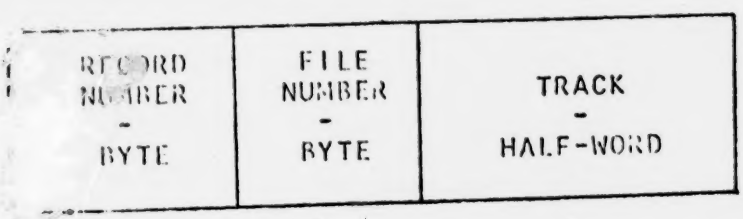


Figure 3.1.1

A Standard System Address

In addition to the physical I/O subroutine READWRIT (which is written in Assembly language), the system contains two subroutines, GETRCRD and PUTRCRD, for reading and writing physical records (not tracks). While special formats are used for various special purpose tracks throughout the system, the record structure handled by these routines is used for all data in the data files and inverted files.

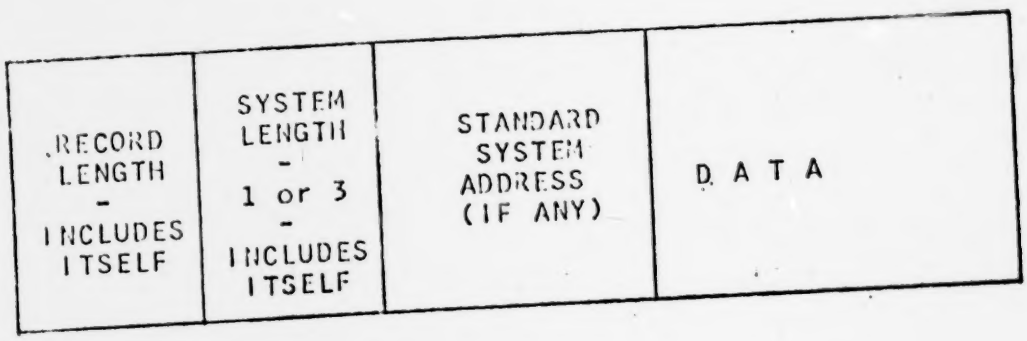


Figure 3.1.2

Physical Data Record

The first half-word is a binary count of the total number of halfwords in the record (it includes itself). The second is the length of the system information (the count includes itself). A value of 3 indicates that the following 2 half-words are a system address of the remaining data of the desired record -- thus a logical record may be a chain of physical records on different physical tracks and so may exceed the physical track length. Naturally the last record in the chain has no chain address.

System File

In order to allow the system to operate with different sets of files, it is necessary to store all the parameters of a given file set within that set of files. This is always done on file 0, NLMO, which is known as the System File. The first track of this file, the System Header record (address 0, 0, 1) contains the names (numbers) of all the files in the system and information about their sizes. The current internal type codes and internal record numbers to be assigned are also stored here (each terminal field in a definition is assigned a unique internal type code or ITC; each record input receives an internal record number or IRN).

Allocation within the system file is handled via information found in the System Header record. The remainder of the system file consists of the IRN Conversion Table, the Definition area and the Name Conversion table. The number of the first track of each table and the number of tracks allocated to each table may be found in the System Header record. These tables are the keys which allow the system to interpret the data stored in the remaining files.

The IRN conversion table is a table consisting of standard system addresses. When a record is input to the system, it is assigned the next available IRN (a three byte, positive binary number). Thus the IRN points, through the conversion table, to the data record directory.

The Definition area contains each of the (up to 25) definitions within the files. Each of the definitions consists of four parts; the Tree Array, ITC Array, Subroutine Arrays (for all 6 phases or system function) and the Name Array. The Tree Array defines the logical structure of the definition. The ITC array associates an Internal

Type Code with terminal data fields, and also delineates repeating fields. The Subroutine Arrays specify, for each terminal field, the routines supplied by the system administrator which should be applied to the data. And, of course, the Name Array contains the names of all fields within a definition.

The final table in the System File is the Name Conversion Table. In it can be found every field name of all the data bases and pointers specifying where and in which definitions they occur.

Consider now the elements that make up an inverted list. Assume we have a record, say record number (IRN)=N, containing the data item NEW (Figure 3.1.3). The inverted list would allow us to look up NEW and the completed search would report that NEW can be found in record N. Suppose the searcher were interested in a record in which NEW MOON occurred as, for example, an idiomatic phrase in a dictionary record describing the word NEW. It is not enough to know that MOON occurs in record N; we also must know in which field it occurs. Thus, part of an inversion in our directories must contain the ITC or internal type code of the field (or fields) in record N under which MOON appears. We are clearly also interested in the concatenation of the two words NEW and MOON. Thus, information needed in an inversion includes: string (or data item); field (or ITC under which the item occurs); record number (IRN of the record containing the desired data); and position within the field (NEW preceding MOON).

Reconstruction is then achieved through concatenation, as indicated in our example.

```
 ENTER COMMAND
*retrieve phrase = new or moon and new moon
       N=V OR V AND V V
         *  *     *   *
 000001  *  =     *   *
 000001  *  *     *   =
 000001  *  *     =   *
 0000001 RECORD HAS BEEN RETRIEVED
 ENTER COMMAND
*print word entry, idiom


 RECORD NUMBER 000006

 DICTIONARY
 ..WORD ENTRY
 ....WORD
       NEW, A.
 ....ORIGIN
       AS. NIWE, NEOWE;CF. D. NIEUW, DAN. AND SW. NY,
       ICE. NYR,GOTH. NIUJIS, L.  NOVUS, GR. NEOS,
       SANS. NAVES
 ..IDIOM
 ....PHRASE
       NEW BIRTH
 ....DEFINITION
       REGENERATION; SPIRITUAL REBIRTH; THE BEGINNING
       OF A RELIGIOUS LIFE
 ....PHRASE
       NEW DEAL
 ....DEFINITION
       THE ECONOMIC AND POLITICAL PRINCIPLES AND
       POLICIES ADOPTED BY  PRESIDENT  FRANKLIN D.
       ROOSEVELT AND HIS ASSOCIATES TO ADVANCE THE
       THE ECONOMIC AND  SOCIAL WELFARE OF THE AMERICAN
       PEOPLE
 ....PHRASE
       NEW MOON
 ....DEFINITION
       THAT PHASE OF THE MOON WHEN IT IS BETWEEN THE
       EARTH AND THE SUN,  WITH THE  DARK SIDE OF ITS
       DISK TOWARD THE EARTH; IT APPEARS AS  A THIN
       CRESCENT  CURVING TOWARD THE RIGHT

 END OF LIST ENCOUNTERED
```

Figure 3.1.3

The Above Example Shows a Retrieval Making Use

of the Concatenation Operator

First, PHRASE was looked up in the Name Conversion Table to determine the correct ITC. Then two lists of records were obtained -- those containing NEW in the desired field and those containing MOON in that field. The concatenation operator then produced a final list -- containing the one record which contained MOON immediately following NEW under the data field PHRASE.

Directory File

There is one Directory File in the system, with 3600 byte tracks. These tracks are of two types; high level or low level directories. They consist of strings (or keys) and system addresses. High level directories point to either high or low level directories. Low level directories point to inverted lists. The Core Directory is at the top of this tree-like directory system and is always found on the first track of file one. Entries in the high level directories are 6 byte keys followed by a 3 byte system address (the record byte is not needed).

All keys are in ascending, logical sort order. Entries in low level directories consist of an entire key (up to 200 characters) followed by one or more Internal Type Code, System Address Pairs. A high level directory key is the last entry on another directory. When a directory fills up, it is split into two directories and the last entry on each directory is inserted in the superior node of the structure.

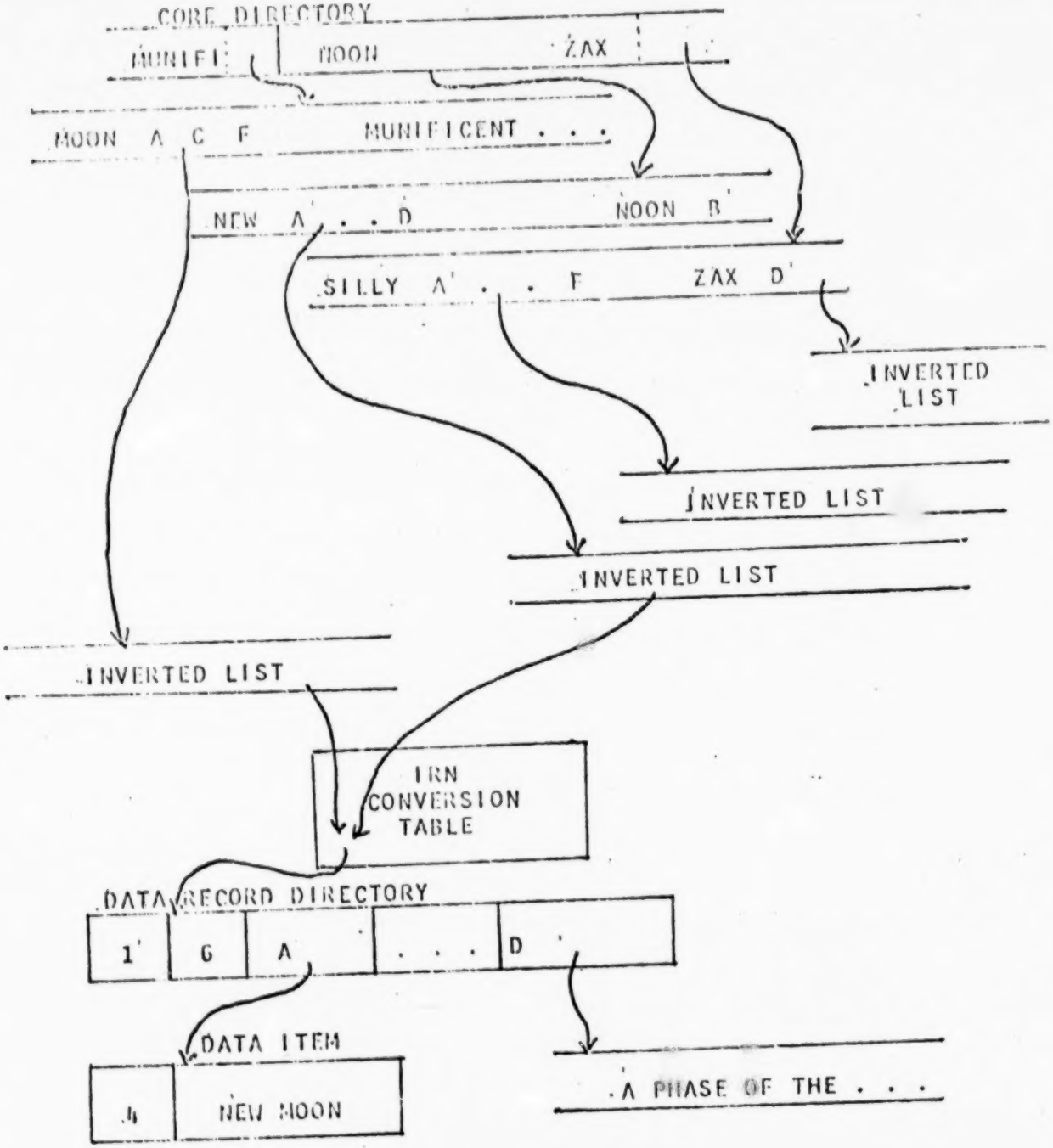


Figure 3.1.4

Directory Structure

Inverted Files

The system allows from one to five inverted list files, consisting of 200 byte tracks. Allocation within these files is handled by the FREESPAC subroutine, whose mechanism is described later. Each list consists of IRN, user byte entries which indicate the record and position within a field for each inversion. The IRN is a three byte quantity, followed by the three user bytes. The first is to indicate sentence number, the second word number within the sentence; the third is unused.

The entire six bytes are used to determine the ascending, logical sort sequence and so the list is ended by an entry (6 bytes) of all one bits. In order to speed access an inverted list (which is in normal physical record format) will not be split over physical tracks unless it exceeds one track in length. Though initially there may be several inverted lists on a track, when one grows too large to fit on a shared track it is moved to its own track.

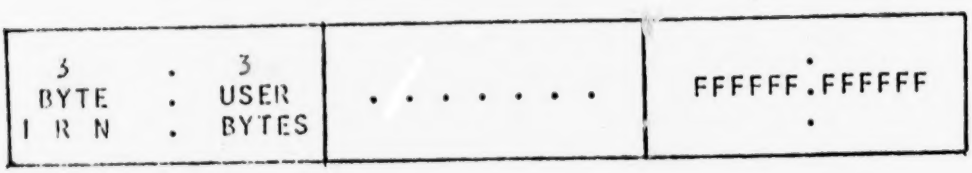


Figure 3.1.5

An Inverted List

Direct Data Files

There may be several, currently up to five, direct data files in the system. It is in these files that a complete data record is stored as an entity rather than as a series of keys in a directory. All data in these files is stored under the format of the physical record previously described. Due to the large differences in numbers and sizes of data fields that the system may handle under different data bases, a chained, rather than continuous, manner of storing data was adopted. Two formats of records exist in these data files: the data record directory and the data items themselves.

The logical entity equivalent to a data record is the data record directory. The IRN points to the data record directory, through the IRN conversion table. It may be several physical records long as it can contain up to 1000 entries. Each of these entries consists of an ITC (Internal Type Code to identify the field), and a system address pointing to the physical record containing that field's data. The ITC occurs in the order delineated by the record's definition; repeating ITC's as well as repeating groups of ITC's may occur.

The data fields are stored in these files as they were passed to the system. As all counts are in halfwords, if a data item contains an odd number of bytes it is padded with a blank (40 Hex). A data field, which can be up to 32,768 half-words long, may be spread over several physical records.

This structure permits greater flexibility in placing the data in the files and cuts down fragmentation within the files. While the chained physical record system would have allowed logical contiguity without

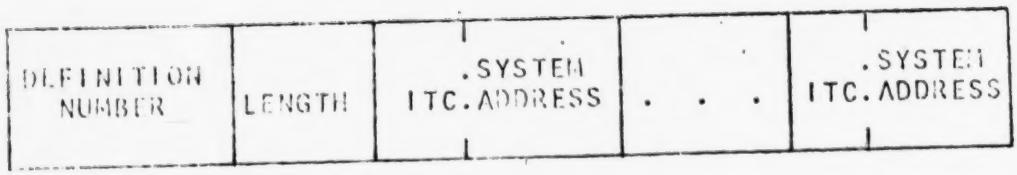


Figure 3.1.6

Data Record Directory

physical contiguity, the data record directory makes possible faster access to selected data items within a record. Allocation of space in these files is done by the FRESPAC subroutine.

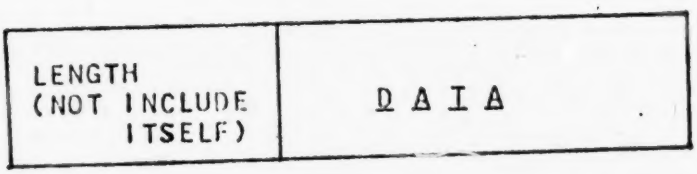


Figure 3.1.7

Data Field

Work Files

The Work Files (from one to three) consist of 2000 byte data tracks. They are used by all phases to store data used within the run that is too large to be stored in virtual memory. All phases are in two parts -- the first in which the user routines are called and temporary files are created. In the second section of a given phase, these files are read and the permanent files modified as indicated by the temporary file.

No fixed format is used for these files -- there are too many different uses. The input function uses them to build a chain of the user's data. Invert allows the user to specify creation of a temporary file of inversions; then this file is read placing the inversions into the permanent file system. The update function uses these files for both of the above purposes. The retrieve phase operators usually result in a third, temporary inverted list. They repeatedly act on two input lists to produce a third output list. The final output list produced is the result or "answer" to the retrieval. The SOLER system will work most efficiently when there are three work files. With two input lists and one output list, it will attempt to keep each of the three in a separate file and thus minimize disk contention and speeding retrieval.

Allocation of work file space is done by the routine WORKSPAC. It grants requests for either cylinders or tracks by maintaining pointers to the next cylinder and track to be allocated. During the input and invert phases requests for space within a file are sequential and so allocation is by tracks. During retrieval requests for each temporary inverted list to be created may be mixed, depending upon the size of each list. Here allocation is by cylinder (they are unlikely to exceed one cylinder), which will reduce disk access time during retrieval.

We are storing a tree structure (our record's information as described in the definition) as a linear string. We may have occurrences of repeating fields to any desired depth -- and with an unspecified number of occurrences. In the accompanying definition we have a repeating field, DEFINITION containing MEANING, PHRASE which also repeats and ORIGIN which does not. Under certain circumstances it may not be possible to determine the proper occurrence number. If we have MEANING, PHRASE, PHRASE; MEANING would be identified as MEANING. But we can have missing data for any field; thus we cannot determine whether or not the second PHRASE is PHRASE, i.e., a second PHRASE related to the given meaning or whether it is really PHRASE, the first PHRASE relating to a missing data item, a second DEFINITION.

We solve this problem by adopting the convention that unless the record structure indicates otherwise, it will be assumed that there is no missing data. Here PHRASE would be correct. If our sequence were MEANING, PHRASE, ORIGIN, PHRASE then the second PHRASE would be correctly identified as PHRASE of a second DEFINITION which is missing. We know this because ORIGIN precedes PHRASE in our record definition. The assumption of no missing data is certainly necessary for the SOLER system to be able to select specified items from an array, and seems quite reasonable.

Allocation Requirements

Until now we have seen some simple methods for allocation of space in three types of system files -- the System File, Directory File and Work Files. The methods have been simple, effective and taken little computer time. These methods were effective because we were dealing with

known data (we specified the contents and format of the data in these files) and in one case, the work files, allocation is not overly critical as the files are clean at the beginning of each run. Finally, in all the above cases, allocation is in terms of whole tracks -- fragmentation is not a problem. Now we are going to have to deal with a quite different method of allocation. This allocation mechanism handles two types of files. In one (the inverted list files) we do know the format of the data but we did not know its character, i.e., will there be many or few, long (several tracks) or short (two or three entry) inverted lists. In the direct data files, we can be dealing with many or few fields -- each field may be long or short and there may be different data bases. Thus, these characteristics may be different in different parts of the files. Another requirement is the packing of data; fragementation can be a problem is you lose 10% of file space, and your files use 10 disk drives. A disk unit is too valuable to discard lightly. Tied in with this requirement is that of machine independence. The files may be on different direct access devices.

To recapitulate, our data allocation mechanism must pack data to reduce fragmentation and wasted space; it must attempt to place related information in one cylinder. It must be data and device independent (for moving head, direct access devices). Finally this mechanism was built before final design of the rest of the SCLER system and it was not known how other parts of the system would use it. It turned out that the invert, input and update functions all use this mechanism in a different manner.

Allocation Mechanism

The allocation mechanism is made up of two main programs and one subroutine. One main program, INITIAL, creates the free space tables -- setting them up to indicate that the while file consists of complete tracks except for the tracks in the tables themselves. These are set up to indicate that they are completely filled with data. The second main program, FIDUMP, may be run at any time to examine the state of the files. This program displays the contents of the tables in a more easily readable format than a simple hexadecimal dump.

The final, and most complex, part is the allocation subroutine, FREESPAC. It is part of the input, invert and interactive (update) functions. That is, it must be used by any function which modifies the inverted list and direct data files. The routine is broken up into four entry points, each with a distinct function. The entry point, SPACINIT, loads the tables of the files into the arrays within the subroutine. TERMIN restores the tables to the files. UPDATE changes the free space table for a track and the effected cylinder. GETSPAC performs the main allocation function, that of determining where the new data entered into the files should best be placed.

3.9 Statistical Formulae Employed

There are many possible statistics that can be constructed to determine the distribution of the occurrences of terms in a document collection. These different measures all attempt to obtain a figure that gives some insight into the possible assignment of the terms to the common, core, or particular category. All statistics are related to the frequency of occurrence distributions of the terms. Let this frequency be f(j,d), the frequency of term j in the d'th review. Then the total frequency of term j in the collection is:

$$F(j) = \sum_{d} f(j,d) \qquad 3\text{-}1$$

The values of f(j,d) can be normalized by dividing by the length of the documents, L(d). Then the normalized frequency is:

$$r(j,d) = f(j,d)/L(d) \qquad 3\text{-}2$$

The value of f(j,d) may be normalized differently by dividing by the logarithm of the length of the document, log(L(d)). Then the log normalized frequency is:

$$\tilde{f}(j,d) = f(j,d)/\log(L(d)) \qquad 3\text{-}3$$

For each of the above distributions, statistics for the occurrence of each term j in the document set were computed as follows:

1) The mean of the values of the distributions.

2) The variance about the mean of the values of the distributions.

3) The third moment about the mean of the values of the distributions.

4) The co-efficient of skewness about the mean of the values of the distributions.

Four of the twelve statistics yielded the best results for the sample set of terms. These were the three applications of the co-efficient of skewness of the distributions and the variance about the mean of the value of the log normalized frequency distribution. Two additional measures are included. They were derived by Don Stone working under the supervision of the author. The derivation of those statistics is presented in Stone's Master's Thesis.[70] The first statistic is similar to one proposed by Sally Dennis.[15] The second is Don Stone's:

5) The variance about the mean of the values of the logarithm of the frequency distribution, normalized by the total frequency of occurrence.

$$\bar{V}(j) = \frac{V(f(j,d)/\log(L(d)))}{F(j)} \qquad 3\text{-}4$$

In an attempt to utilize only the documents where each term appears, the following statistic was derived:[70]

6) The ratio of the total frequency of occurrence of each term divided by the estimator of the Poisson distribution parameter. This gives S(j), the size

Table Structure

These requirements were met by storing all information describing the files in the files themselves. A table structure (rather than one of chains of tracks having similar amounts of available space) was chosen because it describes the location of available free space rather than merely sizes of chunks of empty space. It was important to minimize the size of the tables, both to save file space and save time in accessing the tables (all of which is overhead). The final structure chosen uses between 1 and 2 bytes per track of the file. While this is less than a chain structure (a chain would cost 4 (or 8) bytes per track), it effectively wastes more space because our tables are contiguous and cost several whole tracks.

There are three types of tables needed to describe the space available on a file. The first -- the Parameter table -- describes the file. It contains 3 half-word items; the number of tracks on each, data blocks or cylinders in the file (NDB), the number of tracks on each data block (NIDB) and the internal free space category (IESC). This latter is a constant used in referring to the other tables. All of the information in the Parameter table is also found along with the correct file number in the System Header Record.

Before we go further, we will need some terminology. A complete track is one which has no data stored on it. It is one thousand half-words (2000 bytes) long. Of these, 999 half-words are available for data -- when a track has records in the standard system format on it -- the last record must be followed by a record length count of 0 (for the PUTRCRD and GETRCRD subroutines). We shall refer to a track

containing less than 999 half-words of free space (i.e., already containing one or more records) as a fragmented track.

The second table -- the Cylinder table -- contains one two byte entry for each cylinder in the file. The Ith entry corresponds to the Ith cylinder. The Parameter and Cylinder tables occupy the first track in a file; thus a file can contain up to 997 cylinders. Each of these entries is a binary number--from 0 to 255. The first byte, referred to as Other Free Space (OFS), contains the sum of available space on fragmented tracks of the cylinder. The second, Empty Tracks (EI), is a count of the number of complete tracks in the cylinder. It cannot be greater than 255, thus limiting devices to no more than 255 tracks/ cylinder. Other Free Space can easily be greater than 255; thus the byte, OFS in the table, is actually the other free space divided by a constant, K.

In order to determine the total free space (TFS) in a cylinder, we calculate as follows. The final table, the Track table, contains one byte for each track in the file (the Ith byte is for the Ith track). Each of these bytes is the count of free space (CFS) on the track. This is also greater than 255, so the actual free space is divided by a constant (IFSC, stored in the parameter table). Due to the truncation, it would not be possible to tell a complete track from one which contains a small record -- thus complete tracks are marked by having the CFS byte be FE (hex) rather than having a count.

3.2 The Record Definition Phase

The function of this program is to accept a description of the logical structure of a data record written in an external specification language and translate it into a corresponding internal machine representation in the form of arrays and tables. The specification language of a logical record is described in Backus Normal Form (BNF) in Figure 3.2.1. A record may be inserted into the system from the console or it may be read into the machine from a card disk.

When no syntactic errors are detected in the record definition at input, it will be processed and inserted into the system, thus updating information in the following system tables and arrays: the System Header Record, the Field Name Conversion Table, and the Internal Definition Table, all of which are stored in the System File. However, when syntactic errors, wrong input format or unacceptable ambiguities are detected in the processed record, a proper diagnostic message identifying the nature of the error is printed, the program is aborted and a return is made to the calling program. In this case, a diagnostic flag is set which provides the calling program with the reason for the error return.

Some error checking facilities are built into the program. If a suspicious condition which may not necessarily result in error is detected (e.g., ambiguity which may be resolved by qualification), a warning message is printed and processing continues. However, if an input syntax error is detected, e.g., an invalid (unrecognized) key word, or a subroutine name exceeds eight characters, or a comma is missing, etc., an error (abort) return is taken.

< record definition >::= < name of definition > |
| < record definition > < field description >

< name of definition >::= 001 < b > < field name > < b > ,
< b > < common data flag >

< field description >::= < level number > < b > < field name > < b > ,
< b > < repeat flag > < b > , < b > < user subroutines >

< common data flag >::= ∅ / COMMON-DATA

< level number >::= three digit decimal number where leading
zeros may be replaced by a blank.

< field name >::= any sequence of alphanumerical characters
(including blanks) not exceeding 73 characters in length.

< repeat flag > ::= ∅ / REPEAT

< b >::= ∅ / zero or more blanks

< user subroutines >::= ∅ / < user subroutines > < key word >
< b > = < b > < subroutine name >

< key word >::= INPUT/INVERT/OUTPUT/UPDATE/VALIDATE/RETRIEVAL

< subroutine name >::= any alphanumerical symbol (without blanks)
not exceeding 8 characters in length including the
symbol NONE.

Figure 3.2.1

BNF Description of the Input Syntax

The symbol "∅" here means that the corresponding expression on the left of the ::= may be omitted. The symbol < b > indicates whenever it appears that one or more blanks may (but do not have to) appear at this place.

The NLM Record Definition Program consists of a main portion (RECRDF) and three auxiliary subroutines (HASH, NAMESRCH and INSERT) which compute the hash code of a field name, search and insert elements into the system field name tables, respectively. A functional description (a description of the functional flowchart) of these four subroutines is given in the following four sections.

The HASHNAME Subroutine

This subroutine (entry HASH) computes a hash-code corresponding to a field name which is stored in a binary array called "NAME-FIELD". The hash code is computed by adding the EDCDIC codes for all characters (including blanks) of the field name, and taking the result modules, the number of tracks in the field, the name conversion table which is stored in the parameter N-C-SIZE.

Since the number of tracks in the field name conversion table is a system parameter which may have a different size for different systems, and may change during the life of the system, the parameter N-C-SIZE must be initialized before any hash code may be computed. A special entry print "HASHINIT" has been provided for this purpose.

The NAMESRCH Subroutine

The function of this routine (entry SRCHNAME) is to search a name array (stored in a binary array named SEARCHED-ARRAY) for a name which is stored in NAME-SEARCHED. The structure of the SEARCHED-ARRAY must have the standard format of data structures containing field names.

If the table is not empty, the search is performed as follows. Every name of the table is examined in sequential order (starting with the shortest name at the beginning of the table) until a name is found whose length is the same as the length of NAME-SEARCHED (stored in LENGTH-OF-NAME). Then, both names are matched character-by-character. If there is a match, 'T' is moved to the MATCH-FLAG and the RESULT-POINTER is set to point to the beginning of the matching name before returning. If names do not match, the next name of the same length is located and examined as before. If the end of the table is reached, or if the length of the next name in the table is longer than LENGTH-OF-NAME, the search is terminated. In this case, 'F' is moved into the MATCH-FLAG and the RESULT-POINTER is set to point to that point of the SEARCHED-ARRAY where the searched name would be inserted.

The INSERTEL Subroutine

This subroutine is used to insert some element (stored in an array labeled INSERTED-ELEMENT) into a table called TABLE-TO-INSERT. The pointer, POINTER-TO-INSERT, specifies where the INSERTED-ELEMENT should be inserted.

The insertion is carried out as follows. First room is made in TABLE-TO-INSERT for the INSERTED-ELEMENT by moving all half-words of TABLE-TO-INSERT forward, beginning at the point where the insertion is to be made by an amount equal to LENGTH-OF-ELEMENT. In other words, a window is made for the INSERTED-ELEMENT by shifting the content of TABLE-TO-INSERT to the right beginning at a point specified by POINTER-TO-INSERT. After room has been made, the INSERTED-ELEMENT is transplanted into the cleared slot of TABLE-TO-INSERT, moving one half word at a time. If the

TABLE-TO-INSERT is empty, or if POINTER-TO-INSERT points to the end of this table, there is no need to make room and only the moving operation is performed.

It should be noted that INSERTED-ELEMENT is a 3,600 byte array and that it is not necessary that the inserted element start at the beginning of this array, since its starting position is specified by the POINTER-TO-ELEMENT and its length is given by LENGTH-OF-ELEMENT.

The Main Subroutine - RECRDF

The RECRDF routine is the main data processing unit of the NLM record definition program. During processing it makes use of several auxiliary subroutines, some of which have been described in the preceding three sections. This subroutine is called with two parameters: the DIAGNOSTIC-FLAG and the SYSTEM-HEADER-RECORD. The purpose of the DIAGNOSTIC-FLAG is to convey to the calling program information concerning success or failure of data processing after return from RECRDF. All other information is needed by RECRDF for processing of the input record, since all needed system parameters (addresses to data areas or pointers to system tables, etc.) are contained in the SYSTEM-HEADER-RECORD (SHR). The SHR is the header record of the system file which contains all of the tables and data areas generated by the RECRDF program.

The programming starts at the first paragraph labeled INITIALIZE where the subroutine HASHNAME is initialized by setting its "module divisor" equal to the number of tracks in the field name table. Next, the various pointers, counters and program parameters are set to their initial values and the processing of the input record begins in the paragraph labeled MAIN-LOOP.

The paragraph MAIN-LOOP is the beginning of the main program loop through which the subroutine must cycle during processing of every line of the input record. First the pointers to the INPUT-BUFFER (I-B) and the NAME-ARRAY are set to point to the first character. Then the next line is read from the input media into the I-B by calling the subroutine RDATA. The auxiliary parameter VALU is a diagnostic flag which conveys information about the success or failure of the record operation. If VALU contains a zero, the read operation was successful and a transfer is made to point C-1 where processing continues. When VALU contains a value of 16, an end of file was read at the input media which means that the entire input record has been processed and transfer is made to CHECK-REPEAT-STACK where concluding operations are performed. When VALU contains a number other than 0 or 16, an input read error has occurred; consequently, a proper error message is generated and control is transferred to ERROR RETURN where the error message is printed, the DIAGNOSTIC-FLAG is set to one, and an error return is taken.

When a successful read operation has been carried out, processing of the newly read data starts at C-1. First the level numbers (the first three digits of the input line) are extracted from the input record by performing the CHECK-DIGIT section three times -- once for every digit of the level number. The CHECK-DIGIT section performs a binary to decimal inversion for the byte pointed to by the INDEX-OF-I-B. If something else other than 0, a blank or a decimal number is contained in the field reserved for the level number, a proper error message is printed and an error return is taken. If the level number is zero (or blank), it is assumed that this line is a continuation of the previous field and processing continues at the RESTORE-LEVEL-NUMBER paragraph.

When a position level number has been computed, transfer is made to the point EXTRACT-NAME-FIELD where the name field of this sub-record is extracted, after first removing the leading blanks (if any) and then transplanted into the array labeled NAME-FIELD. If the name field contains several words or symbols separated by blanks, and if the number of blanks between words or symbols is greater than one, all but one of these blanks are removed. After the entire name field has been extracted, a check is made as to whether it is a "Header" and, if so, the HEADER-FLAG in the SHR is set to 'T'. If the number of characters of the name field is odd, a trailing blank is added to make it even because the name field will later be stored in a half-word array which contains two characters per half-word. Next, the has code for this name field is computed by calling the HASHNAME subroutine. Then the address of a field name table track corresponding to this name field is computed by indexing the basic address of the field name table by the hash code of the name field, and the track is read into an array labeled TRACK-OF-FIELD-NAME-TABLE.

At the paragraph CHECK-NAME-TABLE the track of the name table is searched for the field name (using the NAMESRCH subroutine). If the field name already appears in the name table, this name is ambiguous and a diagnostic error message is printed. If the level number is one, the ambiguity is unresolvable; therefore, the program is aborted and error return is taken. Otherwise, a warning message is printed and processing continues as if the name were not already present in the name table at the paragraph NOT-IN-NAME-TABLE. The next step of the program is to check as to whether the field name has appeared in this record before. This is done by searching the name array using the NAMESRCH subroutine. If the field name is again ambiguous, an elaborate check is made as to

whether this ambiguity is resolvable. The ambiguity is not resolvable if both names appear on the same level in the tree of the record structure and have the same predecessor mode(s); that is, the field name(s) of the predecessor mode(s) is (are) the same. The check for unresolvable ambiguity is performed in the TEST-UNRESOLVABLE-AMBIGUITY section by first locating the ambiguous modes, comparing their level numbers and then comparing the predecessor modes when the level numbers are the same. As before when an unresolvable ambiguity is found, the program is aborted after printing the proper error message. If the ambiguity is resolvable, only an additional entry containing a pointer to the tree array is added to the already present name array entry for this field name. If the field name is not in the name array, a name array element for this field name is generated and inserted into the name array. The INSERTEL subroutine is used in both cases to do the inserting.

Finally at the paragraph labeled GENERATE-TREE-ARRAY-ENTRY an entry of the tree array corresponding to this field name is inserted into the proper slot of the TREE-ARRAY. Then the Repeat Stack is checked at the paragraph labeled CHECK-PREV-REPEAT. The Repeat Stack is a linear array whose every entry contains the level number and ITC array index for those fields that have been flagged with the repeat flag (that is, fields that may be repeated). Since repeated fields may be nested within repeated fields, a pushdown stack is required to store the information needed to generate the correct entries in the ITC and subroutine arrays. Whenever a repeated field is encountered, an entry is pushed on top of the stack containing the level number and ITC index of this field. Then for every subsequent field its level number is compared to that stored on the top of the repeat stack. If the level of the present field exceeds the

level number on top of the repeat stack, processing continues at the paragraph INSERT-LEVEL-NUMBER; otherwise, some processing centered about the repeat mechanism must be performed. First, the ITC index of the (previously encountered) repeated field is retrieved from the top of the repeat stack and placed into the next entry of the ITC array; then the name 'SYSBRNCH' is moved into all six corresponding slots of the subroutine array and the repeat stack is popped by decreasing its index by one. Then again, the top element of the repeat stack is checked and similarly processed if necessary. This activity of the repeat mechanism continues until either the repeat stack is empty or until the level number on top of the repeat stack is less than equal to the current level number, at which point normal processing is resumed at the INSERT-LEVEL-NUMBER paragraph when the level number of the currently processed field is inserted into the proper slot of the tree array.

When the name of the field has been disposed of, processing continues at the paragraph labeled GET-NEXT-KEY-WORD where every key word is processed individually. If another key word is found in the input buffer, it is extracted from the buffer and moved into the NAME-FIELD array. If the number of characters in the key word is odd, it is appended a trailing blank by performing the ADJUST-NAME section. Then the table of key words is searched for the key word using the NAMESRCH subroutine If the key word is not found in the table, an error message is printed and an abort return is taken. The table of key words yields a code identifying the key word to the program. If the key word code is between one and six, the key word identifies a user subroutine; in this case transfer is made to the paragraph CHECK-ITC-POINTER.

If the key word code is 7, the key word is a repeat flag. In this case, transfer is made to paragraph SET-REPEAT-FLAG. If the level number is equal to one, the program is aborted because the first level must not be repeated; otherwise, the repeat flag in the tree array is set to 'T' and the repeat mechanism is activated by pushing the current level number and ITC array index on the top of the repeat stack. Then return is made to GET-NEXT-KEY-WORD.

If the key word code is 8, the key word is a common data flag. Again, if level number is 1, the program is aborted; otherwise, the common data flag in the record definition area is set to 'T' and transfer is made to GET-NEXT-KEY-WORD.

The case where the key word identifies a user subroutine is treated at paragraph CHECK-ITC-POINTER. If the ITC pointer in the tree array is found to be equal to zero, it is recognized by the program that a terminal element of the record definition tree has been encountered, and that the first user subroutine for this field is being processed. Therefore, a 'NONE' is initially moved into all six slots of the subroutine array corresponding to this field. If the name of this field is 'Header' (as recognized earlier during processing of the field name), a zero is moved into the corresponding slot of the ITC array; otherwise, the index of the ITC array corresponding to this field is moved into the proper slot of the tree array. Next, at the paragraph GET-NAME-OF-SUBROUTINE the leading blanks are removed and then the subroutine name is transplanted from the input buffer into the proper slot of the subroutine array identified by the code of the previously processed key word contained in the common variable I. If the number of subroutine characters is less

than 8, trailing blanks are added; if the length of the subroutine is greater than 8 characters, only the first 8 characters are used.

When the user subroutine has been moved to the subroutine array, control is transferred to point GET NEXT KEY WORD where the next key word is processed as described in the foregoing. If the entire input line has been processed, control is returned to point MAIN-LOOP where the next input statement is read into the Input-Buffer and the program keeps cycling in this manner until an end of file is read on the input media (VALU = 16 after a read operation).

When an end of file is detected by the program, it is assumed that the entire input record has been processed and a transfer is made to CHECK-REPEAT-STACK. In this paragraph, the repeat stack is checked and if found to be non-empty, the repeat mechanism is activated and repeat entries are inserted into the subroutine and ITC arrays as described in the preceding paragraphs. When the repeat stack has been emptied, transfer is made to UPDATE-SYSTEM-FILE where the data structures generated by the RECRDF program are transplanted into the proper areas of the System File.

First the number of this definition is computed. Then the field name table is updated as follows. For every element of the name array, an element of the field name conversion table is assembled in the proper format. Then the hash code for the field name of this element is computed by the HASHCODE subroutine. This hash code is added to the base address of the field name table, thus yielding the address of the field name table track for this element. The track is read in and searched. If the field name for this element does not yet appear in the table, the entire element is inserted using the INSERT subroutine; however, if the

name is already in the table, only the indexes are inserted into the table (simply by changing a pointer to point to the indexes rather than to the beginning of the element before calling the INSERTEL subroutine). Then the updated track is written out on disk and the next element of the name array is processed in the same fashion by returning to TRANSPLANT-NEXT-NAME. This cycling continues until the entire name array is processed, when transfer is made to MOVE-RECORD-DEFINITION where the individual data structures of the record definition are strung together in the correct order and written out on disk.

However, before the tree array can be written out on disk, it is necessary to compute all the name array pointers in the tree array. This is done by scanning all elements of the name array in sequential order and inserting a pointer to the name array element into the name array pointer of every tree array element pointed to by the indexes of the name array element. When the tree array is updated (the entire name array is scanned), a dummy entry is appended to the end of the tree array, and its length is computed. If the length of the tree array is less than one track (3600 bytes), the ITC array is appended to the tree array; then the subroutine arrays are appended and then the name array. After transplanting a half-word, a check is always made as to whether the track has been filled (1800 half-words) and, if full, it is written out on disk and pointers are reset so as to start loading a new track at the beginning. This process continues until all elements of the record definition have been written out on consecutive tracks into the system file in this order: tree array, ITC array, subroutine arrays and the name array. Then the system header record is updated and written out, the diagnostic flag is set to zero (normal return) and an exit from the subroutine is taken.

3.3 Input Phase

The purpose of the input phase is to accept raw data and store it in the data base. For each type of data to be processed, there is a definition which describes the logical structure of a record.

For each field in the record there is a corresponding terminal field in the definition. Each terminal field in the definition has associated with it an Internal Type Code. If a group of fields is to be repeated, the field immediately following the group of fields has as its user subroutine name the name of the system branch routine. The Internal Type Code associated with this field is a pointer to the first field of the group of fields to be repeated. This will be more fully discussed later.

It is the job of the user subroutines to read the raw data, whether it be on cards, tape, etc., perform any formatting required and pass the data to the input phase in the order prescribed by the definition.

Since the system has been designed to handle data in many different forms, the user subroutines are designed to handle the unique characteristics of each type of data and thus provide the interface between the raw data and the input phase.

The data is permanently stored in the data files of the data base. The structure of the stored data will be discussed later.

3.3.1 Operation of the Input Phase

The program NPUTEXEC performs all initialization procedures required by the input phase. This routine accepts as input the phase name and the name of the definition required for processing the type of data currently under consideration.

For example:

< Input Toxicology Information File >

as an input command calls for the definition of the record structure of the Toxicology Information File to be processed by the input phase.

Control is then passed to the main input routine - INPUT. INPUT calls BUFRMNG to convert the definition name in the input command to a definition number and then calls DEFCHAR to retrieve the logical structure of the definition currently being used.

NICEDEF is then called to build the Fetched-Definition-List and User-Subroutine-List. The information in these lists include the name of the entry point of the user subroutine to be used to process the data for each terminal field in the definition, the inhibit flag for each terminal field, etc.

The next step in the processing of data is to obtain an internal record number (IRN), fromIRNWORK, to be used in referring to the record being processed.

At this point, processing on the data is begun. The name of the entry point of the user subroutine required to operate on the data for each terminal field in the definition is obtained from the User-Subroutine-List and passed to CALLSUB along with the Internal-Type-Code (ITC) of the field the definition requires data for, and other parameters. CALLSUB initializes the loading of the user subroutine and passes control to it.

The user subroutine then reads the raw data for a field and determines if the ITC of the field of data read is the same as the ITC of the field required by the definition. If not, the user subroutine sets the value of Returned-Length to ∅ and returns control to INPUT. INPUT then sets the value of the length of data for the ITC required by the definition to ∅, determines the next ITC required by the definition and passes control back to the user subroutine via CALLSUB.

This process continues until the ITC of the data read matches the ITC required by the definition. At this point the user subroutine moves the data read into the Immediate-Return-Buffer, sets the value of Returned-Length to the number of halfwords in the Immediate-Return-Buffer and passes control to INPUT which sets the value of the system branch flag for the field to 'Y'.

The Immediate-Return-Buffer is 1∅∅∅ halfwords in size. If the length of data for a given field is greater than 1∅∅∅ halfwords, then the user subroutine sets the value of the Overflow-Flag in the System-Data-Area to 'Y' and returns to INPUT.

When the Overflow-Flag is on, INPUT empties the Immediate-Return-Buffer and returns control to the user subroutine. This process continues until there is no more data for a given field.

This continues until all the data for a record has been read through the user subroutine.

If at any point the name of the user subroutine entry point in the User-Subroutine-List is the name of the system branch routine, the following algorithm is applied.

1) The ITC of the field with the system branch routine as the name of its user subroutine entry point is a pointer to the first field of a group of fields in the definition to be repeated.

2) A check is made to see if any of the fields between the field pointed to by the ITC in 1 and the field with this ITC had any data associated with them (i.e., their system branch flag was set to 'Y').

3) If the result of (2) is positive, then Input resumes processing from the field pointed to by the ITC in (1).

4) If the result of (2) is negative then processing continues from the point where it stopped.

Input keeps a list of ITCs had the number of halfwords of data returned for each. After each return of the user subroutine, INPUT calls LONGSTOR to temporarily store the data in the work files.

The data is temporarily stored in a chained manner. As the data for a record accumulates to more than a track (1000 halfwords) a new track in the work file is used with the address of the previous track stored on the new track.

Thus when the data for a record has all been temporarily stored, we have a list of ITCs, the length of data for each, and the address of the last track used for temporary storage. The data on the last track used contains the address of the next to the last track used, etc.

When all the data for a record has been temporarily stored, control is passed to DATARECD which initiates the procedures for permanently storing data in the data base.

An allocation of space is obtained from FREESPAC and the data for each field in the record is stored on disc in the following manner;

Length of Record	Length of System	Overflow Address (if any)	Logical Length of Data	Data
< 1/2 word >	<1/2 word >	<full word>	< 1/2 word >	

Where "Length of Record" is the physical length of data for a field stored in this entry, "Length of System" is 1 if there is no overflow address and 3 if there is an overflow address. There is an overflow address if it is not possible to store all the data for a field in one entry. The overflow address is the address of the remainder of the data for the field.

There is also a directory for each record stored in the data base. The directory contains a pointer to the data for each ITC in the definition that had data associated with it. The directory is of the following form;

Length of Record	Length of System	Overflow Address (if any)	Definition Number	Length of Directory	ITC	Pointer	ITC	Pointer	...

When the data has been permanently stored, IRNWORK is called to store the address of the directory of the record processed and the IRN associated with it.

An entry is then made on the .INPUT. list, of the definition number used, for processing by the invert phase.

If there is another record to be processed under the same definition, INPUT resets all pointers, counters, and flags and repeats the processing procedures.

If there are no more records to be processed under the same definition, control is passed to INPUTEXEC. At this point, either a

new input command may be given for processing data under a different definition or processing may be terminated.

In the later case NPUTEXEC performs the required termination procedures such as writing out the System-Header-Record..

3.4 The Update Phase

Updating is an integral part of information systems. Erroneous data which enters the files must be corrected. Correct data must be kept current, and may finally become obsolete. The update function of our system provides for all of these functions. While the update phase may be used from a batch mode, as may all phases of the system, the design emphasis has been toward interaction with a terminal user. Its responsibility is to change the data base, as directed by the user, and to maintain correspondence between the directory structure and the data. There are two major systems to be updated -- the direct data structure and the directory, inverted list structure. For reasons of expediency, directory update is accomplished by placing an inverted list entry of zeroes over the element to be deleted. Though inverted list entries are in sort order, our sorts could easily be modified to ignore a zero entry. Thus the directory update is quick.

A more careful approach was chosen for updating the direct data files. The operation of deleting data is relatively simple -- a dummy record is written where the original data field existed -- this single half-word -- record length count is needed in order to maintain the correct record addresses between data record directories and other data items on the track. This decision, to update the direct data files as the update progressed, was quite important for modifying data. Modification means deletion of the old data. Items are deleted, the free space tables are updated, and so the new data items can replace the old, thus keeping data items within one cylinder.

During the invert function, the system inverts each record under the field names of the data field which contain data (using a special ITC of zero). This allows the user to specify RETRIEVE CATEGORY = FIELD NAME and so obtain all the records containing data for a specific field. When ADD or DELETE commands are processed, these inversions may change and it is part of Update's task to maintain consistency here. Thus, update must modify and maintain consistency between three basic items -- data, data inversions, and finally field name inversions.

General Description of the Update Mechanism

The update system is part of what we term the "interactive" phase. This phase includes retrieval, output and update functions. Hence, we can select records to be updated using the full selection power of the retrieval mechanism. One merely retrieves the desired records; they are placed on the "active" list. Then one can perform one of several commands -- PRINT, LIST, DELETE, MODIFY or ADD (though other commands are possible, they are retrieval commands and affect rather than interrogate the "active" list).

The operation of an update command is divided into two sections; the user section and the system section. This division allows user routines to veto deletion of key fields and maintain the integrity of the data base should a user routine result in abnormal termination. During the user section, data and other information is exchanged between the user routines and the system; and various temporary files (within the data set, the Work file) are built. During this section, user routines from several phases come into use (we are placing data into a state for full retrieval) and must execute the user validate and invert routines, as

well as the update and output phase routines. Naturally, which routines are executed depends upon the exact command.

The syntax of the various commands requires that they specify certain fields. If none are given, the entire record definition (all fields) is assumed. The desired command is processed repetitively against each record of the "active" list. That is, each field is processed for each record of the list (provided the record contains the specified field or fields). When all records have been treated, the user may enter a new command.

A system sub-executive (comprised of the routines PRINTA, PRINTB, OUTPUTPH, and DRDSCAN) is responsible for interpreting commands. It builds a list of the ITC's to be processed, and using this list as a guide, matches the data record directory of the current record against the record definition, and taking subscripts into account, feeds the data record directory and a pointer to the correct ITC, address pair entry into the update-output mechanism.

The update phase contains its own executive, a lower level sub-executive under the retrieval and output executives. This routine (MINIEXEC) actually retrieves the data for the desired field, and if the print flag is on, sends it to the user's output routine. For a print command, no other action is taken. If the command was DELETE, ADD or MODIFY the data item and the pointer into the data record directory are sent to the appropriate routine. This routine will store information concerning the field in a table, create temporary files and feed the data item to the correct user routines. When this has been done for each field, the user section is completed.

```
 ENTER COMMAND
*retrieve name = chris
       N=V
        *
 000000 =
 000001 =
 000001 =
 000001 =
 0000001 RECORD HAS BEEN RETRIEVED
 ENTER COMMAND
*delete name, address

 RECORD NUMBER 000001

 PHONE DIRECTORY
 ..NAME
 ....LAST NAME
       NEW
      FIELD HAS BEEN DELETED
 ....FIRST NAME
       CHRIS
      FIELD HAS BEEN DELETED
 ..ADDRESS
 ....STREET
       252 E 88 ST.
      FIELD HAS BEEN DELETED
 ....CITY
       NEW YORK
      FIELD HAS BEEN DELETED
 ....STATE
       NEW YORK
      FIELD HAS BEEN DELETED
 ....ZIP CODE
       10017
      FIELD HAS BEEN DELETED


 END OF LIST ENCOUNTERED
 ENTER COMMAND
*retrieve name = chris or state = new york
       N=V OR N=V V
         *  *   * *
 000000 =  *   * *
 000000 =  *   * *
 000000 =  *   * *
 000000 =  *   * *
 000000 *  *   * =
 000000 *  =   * *
 NO RECORDS SATISFY THIS RETRIEVE
```

Figure 3.4.1

Example of a Delete Command

For a print command, the output executive will have completed processing the record, and will go on to the next record. For an update command, it now calls the unload entry point. This entry point (MINIEND) determines the correct command and calls the appropriate routine. In this routine, the tables (containing an entry for each field) will be scanned and fields added or deleted. Temporary files will be read and inversions added or deleted. Finally, the data record directory will be updated (or deleted) and the update for the record will be complete.

Update Subroutines

The Update phase is invoked by three commands: DELETE, MODIFY and ADD. Below, we give a brief description of the subroutines used by each command.

(a) DELETE

A logical flowchart of the DELETE operation is shown in Figure 3.4.2. Subroutine MINIEXEC is entered once for each field and is also used to process a print command. It fetches the data and if necessary feeds it to the user's output routine. Before returning, it calls DELETE.

The DELETE subroutine is called for each field and stores in a table information describing the field to be deleted. It then passes the data to the user invert routine which calls CREATE. This creates a temporary file of the original data inversions. Finally, the data is fed to the user update routine.

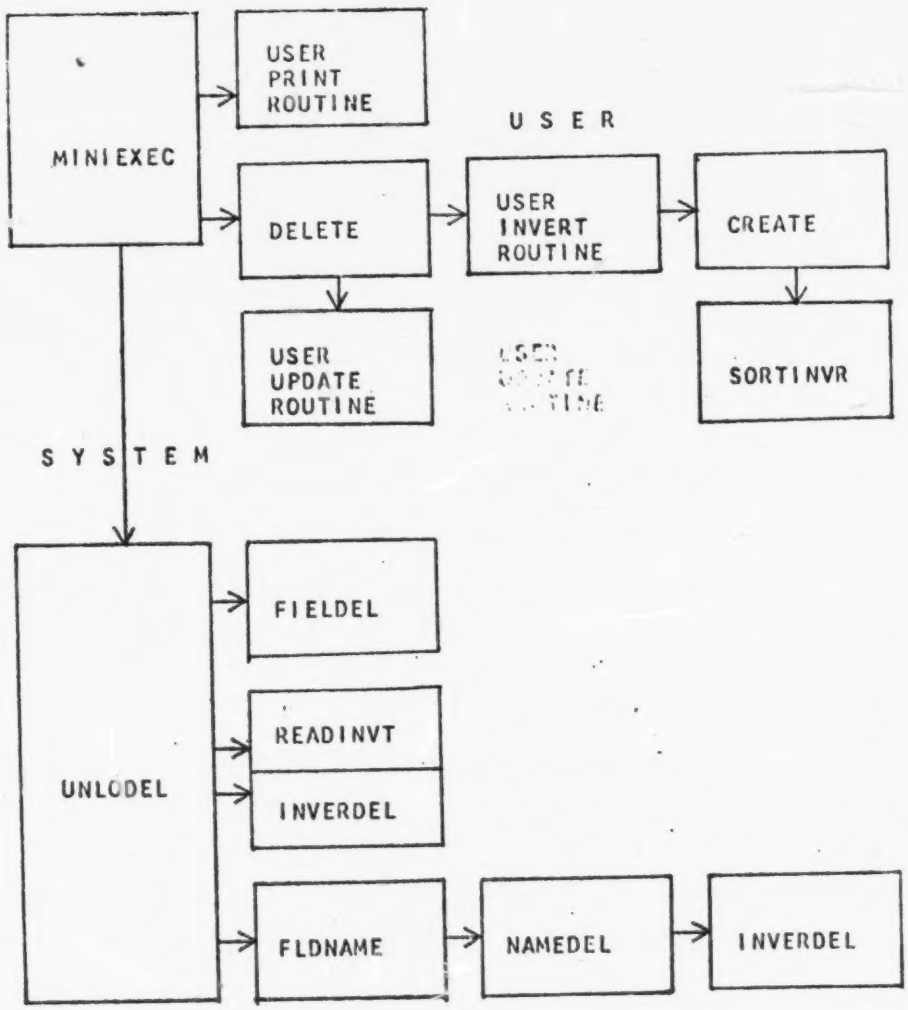


Figure 3.4.2

Logical Flow of a Delete Command

UNLODEL supervises the unloading of the tables and files for a delete command. It must update the data, the data inversions and finally the field name inversion. First, the data fields are deleted. The table of delete elements is sorted, and then each element is processed against the data record directory.

FIELDEL is called for each record in the delete table and writes a dummy record to delete the data field.

UNLODEL then calls READINVI to obtain each entry of the inversion file, and then passes this entry to INVERDEL.

We have previously created two arrays of pointers into the name tables, one from both the original and updated data record directories. These are sorted and processed together. Entries missing from the updated array represent deleted field name inversions and are processed by NAMEDEL.

(b) MODIFY

The MODIFY command is implemented according to the logical flowchart in Figure 3.4.3. The calls to MINIEXEC are exactly as they were for DELETE. However, after fetching the data instead of calling DELETE, the data is passed to CREATMOD.

CREATMOD is called for each field; it stores information in a table and feeds the old data to the user invert routine to create a file of old inversions. The user update routine returns the modified data and the invert routine is called again to create a file of new inversions. The data is stored in a temporary file by LONGSTORE.

(c) ADD

Figure 3.4.4 illustrates the logical flow of ADD. In general, this command is very much the same as a delete. We will add instead of deleting field name inversions; we will add data inversion and will add

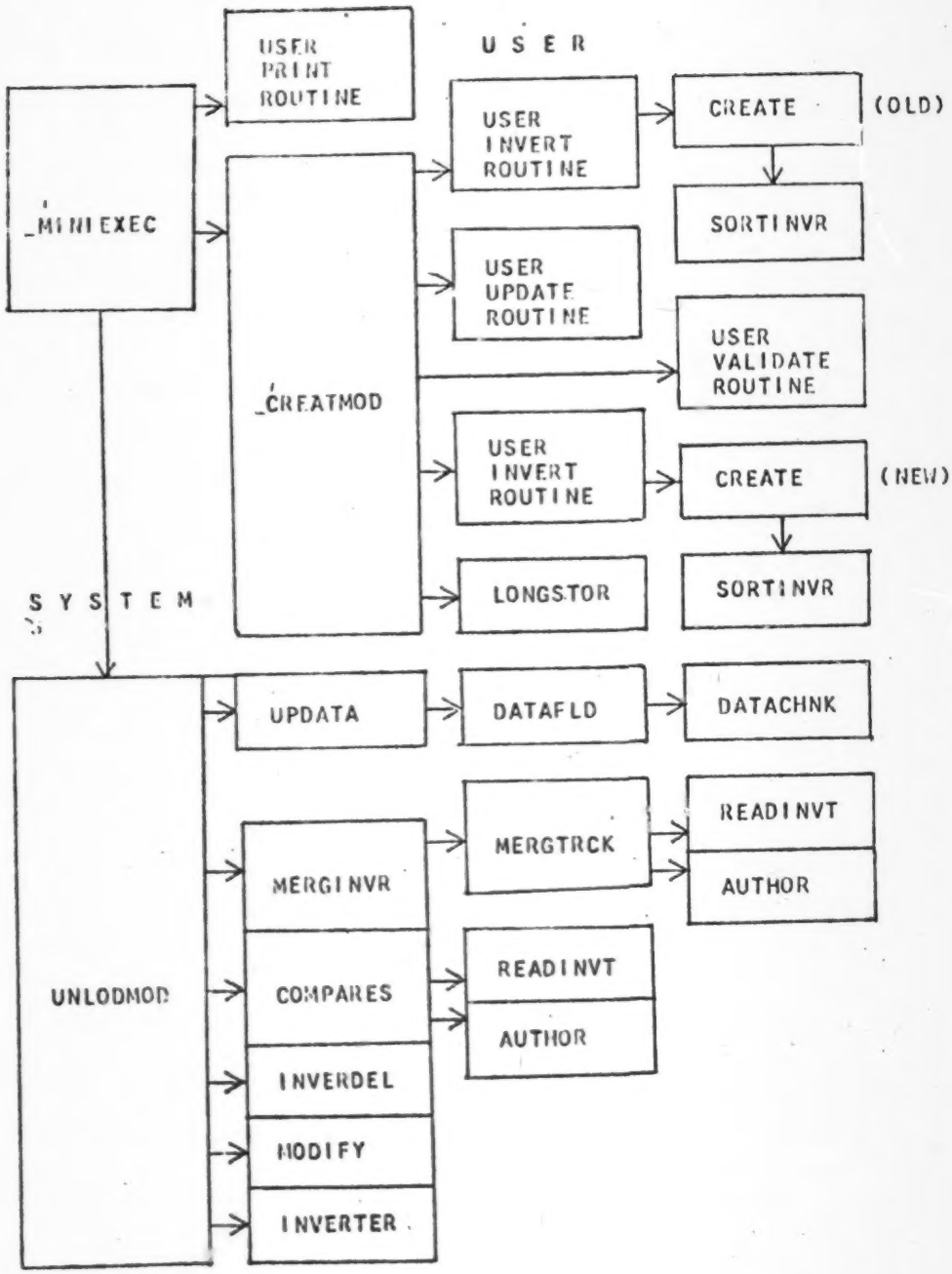


Figure 3.4.3

Logical Flow of a Modify Command

data using the routines from the MODIFY command. CREATMOD is called to obtain the data to be added. The data is fed to the user invert routine and then to LONGSTORE to create the temporary inversion and data files.

In UNLODADD, UPDATA first is called to place the new data into the permanent direct data files. The new ITC, ADDRESS pairs are then placed in the data record directory.

The field name inversions are processed as before by FLDNAME. Now, however, NAMEDEL calls INVERTER to add the new inversions. The inversion entries are read by READINVI and the system routine INVERTER used to add them to the inverted lists. This time the new array is the first parameter (it will have the extra entries; as data fields have been added, not deleted). Now the update is UPDATA.

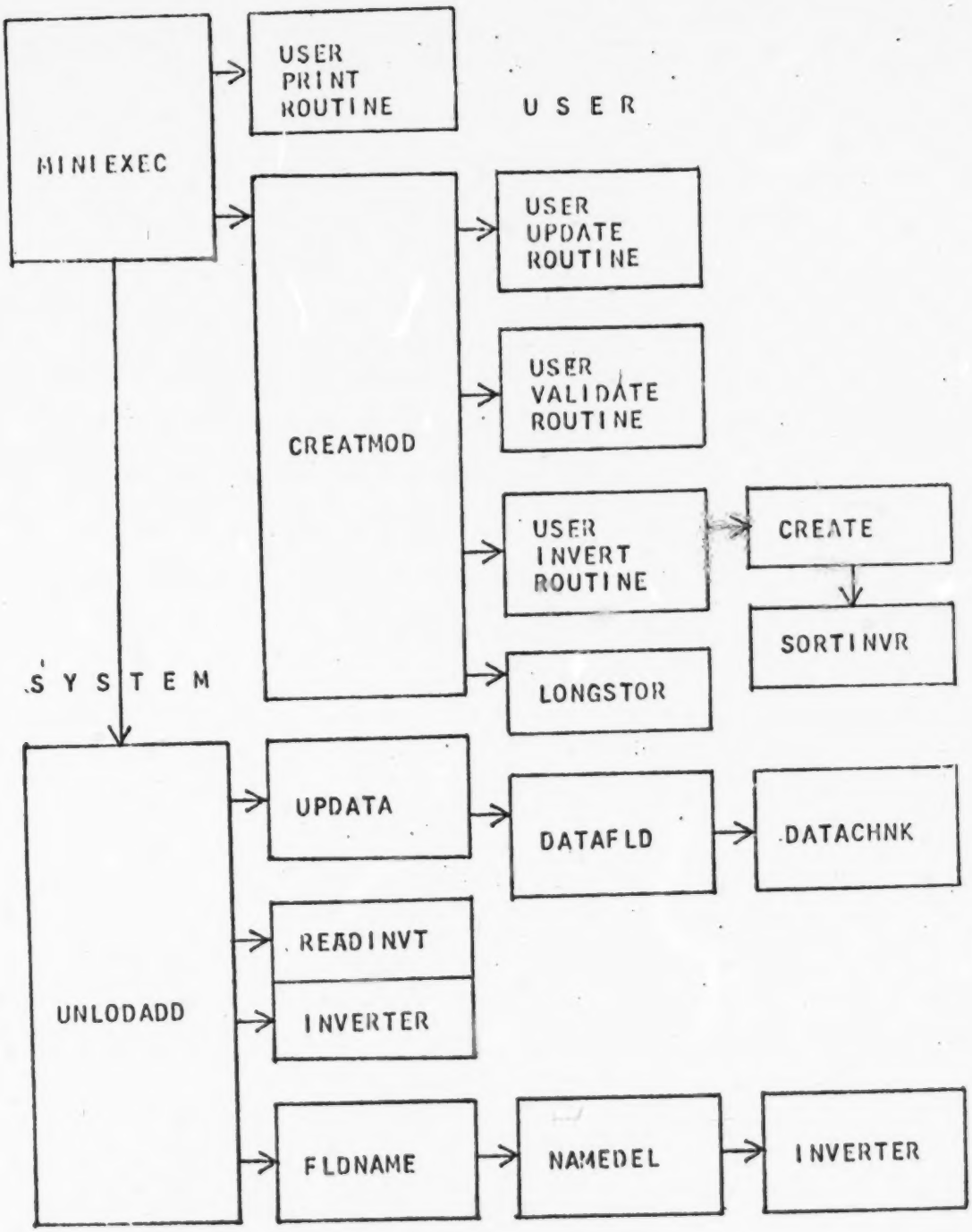


Figure 3.4.4

Logical Flow of an Add Command

3.5 The Invert Phase

In any information retrieval system implemented on a computer, there must be a mechanism by which the system can efficiently access information desired by the user. The invert phase is a processor which forms inverted lists from the records input to the system. These lists are used by the retrieve phase to retrieve records that satisfy a request.

The invert processor has been designed and implemented to make efficient use of disk space and to minimize retrieve time. The former is done by avoiding unnecessary fragmentation of inverted lists on tracks by locating a track with enough space to accommodate the inverted list segment rather than using the first available track. Thus, a segment of an inverted list which is equal to or less than one track in length will not be broken up so as to reside on several different tracks. This technique also minimizes retrieve time, since the number of disk accesses is reduced. Another scheme used to optimize response time is avoiding unnecessary fragmentation of an inverted list on cylinders, which reduces movement of the read-write heads.

Another major design consideration is error recovery capability. This facility is used when the length of a string in the data to be inverted is greater than the maximum length of a string which can be accepted by the system directory. Under such a condition the phase stops the processing of the current record, but instead of completely terminating, continues processing with the next record on a list of records to be inverted.

Finally, the phase can be initiated from either a terminal or card reader. This gives a user the flexibility of using the more appropriate means.

The invert phase takes as inputs a definition name, a definition, a directory and inverted lists, and updates the latter two. The invert processor consists of several subprocessors as shown in Figure 3.5.1.

1. Accepting the definition name and converting it to a definition number.

2. Input of the definition and user subroutine table for the records to be inverted.

3. Input of the appropriate IRN list.

4. Processing of the IRN list which consists of input of data directory, processing values of terminal fields, processing field names, and formation of inverted lists.

Figure 3.5.1

Subprocessors of Invert Phase

At the initiation of the invert phase INVREXEC accepts the definition name of those records input to the system but not yet inverted. The definition name is converted into a definition number. If no such definition name is found, the processor terminates. The conversion is performed by MAININV.

The definition corresponding to thedefinition number found above is read into core by MAININV. The definition includes the name array, tree array and ITC array. These structures play an important role in the processing of field names.

The user subroutine table associated with the definition number is read into core by MAININV. This table specifies a user subroutine for each terminal field in a record. Each entry in this structure is an ordered quadruple which consists of an ITC (Internal Type Code), a name of a subroutine which is to be applied to the data associated with the ITC, a sysbranch flag, and an inhibit flag. The ITC is a positive integer which specifies a terminal field in a record. There is a unique mapping of each terminal data field in a record into the positive integer. This assignment is made in the definition phase. The sysbranch flag is used to process repeated fields, while the inhibit flag, when set, prevents execution of the subroutine. A user subroutine table is as shown in Figure 3.5.2.

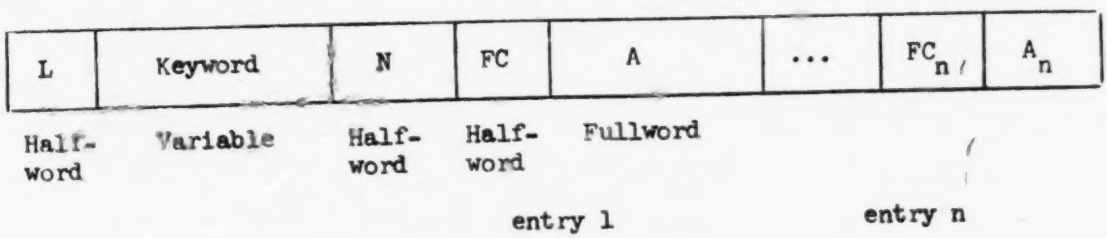
In order to understand how the list of record numbers to be processed are located, it is necessary to give some explanation of the structure of the system directory which is used to access inverted lists. The directory is a tree structure. Each terminal node is an

ITC	Subroutine Name	Sysbranch Flag	Inhibit Flag
ITC_1	Subroutine $Name_1$	N	N
ITC_n	Subroutine $Name_n$	N	N

Figure 3.5.2

User Subroutine Table

entry in the directory; each non-terminal node is used by searching routines to locate an entry. The format of an entry is shown in Figure 3.5.3.



L is the length of a keyword in halfwords
Keyword is a string
N is the number of entries
FC_i (i = 1,2,...,n) is a function code
A_i (i = 1,2,...,n) is an address

Figure 3.5.3

Structure of a Directory Entry

Entries can be partitioned into two classes. Class I is the class of strings which starts and ends with a ".". Each function code in this entry is a definition number. There is a unique mapping of the system definition names into the positive integers as each definition is entered during the definition phase.

Class II entries are all other entries. Each function code in this type of entry is an Internal Type Code.

The address associated with each function code points to an inverted list of record numbers. All addresses in an entry of the type Class I point to a list of record numbers to be inverted. All addresses in an entry of the type Class II point to a list of record numbers which have been inverted.

An address consists of a record number, file number, and track number. The record number specifies which record on the track is to be located; the file number specifies the file to be accessed, and the track number specifies which track in the file is to be accessed. The address format is shown in Figure 3.5.4.

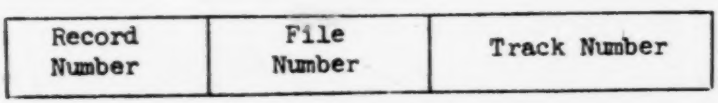


Figure 3.5.4

Address Format

The list of record numbers associated with the definition number found above must be located and read into core. SETTABLE begins this process by accessing the input entry in the directory. The addresses

in this entry point to a list of internal record numbers to be inverted. The relationship between the input directory entry and the IRN lists is shown in Figure 3.5.5.

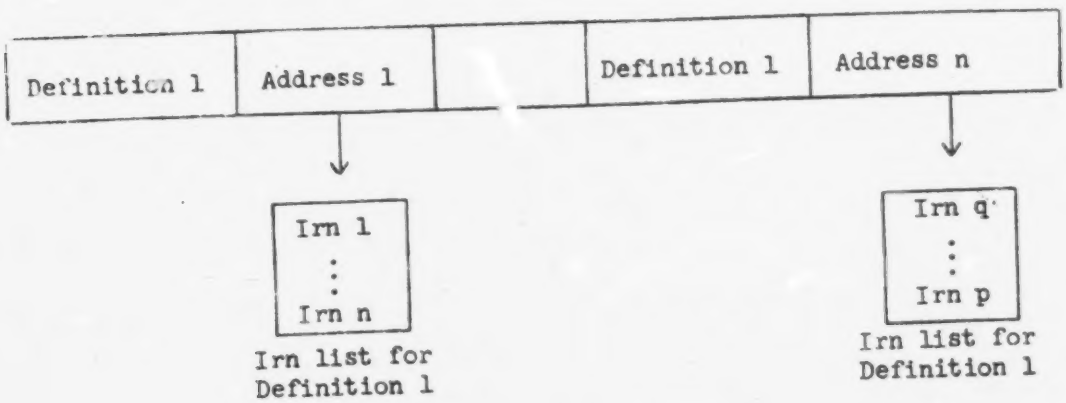


Figure 3.5.5

Relationship Between Input Entry and IRN lists

The input entry is searched by SETTABLE in order to locate the definition number found in step 1. The address associated with the number is used to read the IRN list into core.

Processing of an IRN in the IRN list is divided into four parts. The first part is the conversion of the IRN into an address by DIRECTOR. The address is used to read a data directory into core. This structure, which is created during the input phase, specifies the sequence of terminal fields that appear in the record. Each entry in the data directory consists of an ITC and an address. The ITC identifies a field in the record; the address is a pointer to the data associated with the ITC.

Since each entry in the data directory specifies a terminal field and where it is located, the value of all terminal fields in the record are processed by having DIRECTOR step through each entry in the data directory. By using the address in an entry, the data associated with the ITC of the entry in question is read into core. For each string in the data, the length of the string and the user bytes are calculated. The user bytes indicate the word and the sentence in which the string appears. The string, its length, the user bytes, the ITC currently being processed, and the internal record number are entered as an ordered quadruple into a temporary inverted file by TEMPBUF. This file prevents a record containing an error to be partially inverted.

At the very beginning of the processing of an IRN, a permanent name index array is cleared. Each time a terminal data field is processed (i.e., an ITC), an index to the name of this terminal field and indices to the names of any fields superior to the current field are entered into a temporary array by the routine SUPERIOR. The elements in this array not in the permanent name index array, are added to the permanent name index array. As a result, the permanent array will contain indices to all those field names for which data appeared in the record after all terminal fields have been processed.

At this point, the permanent array of name indices are ready to be processed by DIRECTOR. Each index is used to locate the name of a field in the name array. For each string in the field name, the length of the string and the user bytes are calculated. These three quantities as well as the ITC and the internal record number of the present record are

entered into the temporary inverted file. The ITC for a string in a field name is zero in order to distinguish a field name string from a data string.

The last step is unblocking the temporary inverted files in order to form the permanent inverted lists. The unloading is accomplished by inputting each ordered quintuple in the temporary file to the "Invert" subroutine which forms the permanent lists. The quintuple is a string, its length, function code (ITC), user bytes, and internal record number. The invert subroutine searches the directory for the string and function code. If a match is found, the internal record number and user bytes are added to the inverted list associated with string and function code. Otherwise, the string and function code are added as an entry in the directory and an inverted list consisting of the internal record number and user bytes associated with the new entry is created.

3.6 The Retrieval Mechanism

The SOLER retrieval mechanism is designed to operate interactively in a time-shared environment. The searcher is provided with powerful retrieval tools to aid him in satisfying his requests.

One of the advantages of the system is that the actual data in the data base is never accessed during retrieval. The directory and inverted list structures allow full text search without direct data access. The previous sections of Chapter 3 have described the internal structures needed to understand the logic of the retrieval phase; this section explains the logic itself.

The interactive phase of SOLER is divided into the retrieval and output mechanisms. The main purpose of the retrieval mechanism is to create an active list of records which result from the user's request; the purpose of the output mechanism is to print data from records in the active list. Although the two mechanisms are bound together into one program, the user executes them by separate commands (see SOLER User's Manual, Appendix A).

The user of a general-purpose interactive retrieval system is interested in both the ability to make varied types of requests and the speed with which these requests are processed. In general, these two considerations conflict with each other. Dodd, in his discussion of inverted list systems[1], states "The virtue of such a system is that it allows access to all data with equal ease. Consequently, it is more suitable for situations where the data retrieval requirements are less predictable,... Although the inverted list approach lends itself to easy retrieval, storing and updating data is more difficult, because of the

maintenance of the large dictionaries." With this in mind, the SOLER retrieval mechanism was designed to employ an inverted list structure that provides the user with a very powerful tool which responds within a matter of seconds.

There are five types of commands which comprise the retrieval mechanism: the browsing commands, the data base limitation commands, the result-saving commands, the miscellaneous commands, and, finally, the retrieval commands.

The browsing commands allow the user to directly investigate the data base. By executing a DESCRIBE command, the user receives a description of the files which comprise the data base. Because retrieval and browsing depend on categories of data, this facility is useful as a means of learning the structure of the data. The WHERE command locates all of the occurrences of a specified value in the data base. This command provides the user with the opportunity to find all contexts in which a term occurs. The AROUND, BETWEEN, and TRUNCATE commands provide a direct view into the SOLER directories. AROUND finds values in an alphabetic neighborhood; BETWEEN finds values within upper and lower limits; and TRUNCATE finds values which begin with the same set of characters. Since these three commands search for values within a category, a user can, for example, browse through names in a phone directory or synonyms in a dictionary.

The data base limitation commands allow the user to focus his attention on a subset of the data base. The QUALIFY command accepts a list of category names; until the qualification is removed, all searching is done only within the categories specified for qualification. By

issuing a RESTRICT command, the user can limit his retrieval to a single set of records in the data base until the restriction is removed.

The result-saving commands facilitate browsing without the necessity of re-retrieval. The active list resulting from any retrieval command can be temporarily stored by the SAVE command. At any later time (in the same session), the RESTORE command reactivates the stored list or the ERASE command destroys it. The user can retrace his processing by effectuve use of these commands.

The miscellaneous commands are the links between the user and the administration of SOLER. The COMMENT command saves the user's remarks for the SOLER administrator to read at a later time. The SET command controls various settings of SOLER conditions. In addition to the settings applicable to the output mechanism, the SET command controls the addition of temporary logical operators, tracing of retrieval operations, special symbols used in commands, and capture of the SOLER-user dialogue in a cataloged file. The END command terminates the SOLER session.

The set of retrieval commands is the basic feature of the retrieval mechanism. In order to create an active list containing the results of his query, the user must issue a RETRIEVE command composed of logically connected clauses of search conditions. The logical connectives used to join these clauses are called "level-1 operators". The standard SOLER level-1 operators are "AND", "OR", and "ANDNOT", which are the usual logical connectives. Each clause is composed of a category name and a set of logically connected terms. The logical connectives used to join these terms are called "level-2 operators". The standard SOLER level-2 operators are "AND", "OR", "ANDNOT", "CONCAT", and "SEQUENCE". The

"CONCAT" operator is the space between terms in a phrase; it selects adjacent terms from the data. The "SEQUENCE" operator (denoted by ":") selects terms which occur in sequence (not necessarily adjacent) in the same sentence of the data.

Initially, the command is interpreted and coded in an internal form. Then, each clause is evaluated by finding the inverted list for every term in the category specified in the clause, and performing the level-2 operations on these lists. The result is one generated list for each clause in the command. Finally, the level-1 operations are performed on these generated lists to produce a final list, which is the active list.

The APPLY command has the same format as the RETRIEVE command with the exception that the first clause is prefixed with a level-1 operator. The processing is the same until the final step, when the final list and the previously active list are used as the operands for the additional level-1 operator. The result of this additional operation is the active list. In this way, new conditions can be specified to be applied to the active list. The REPEAT command takes the internally coded form of the previous RETRIEVE or APPLY command and, without having to search the directories again, reprocesses the logical operations. This command is particularly useful when the previous retrieval is to be reprocessed after a restriction has been removed or after the retrieval tracing facility has been turned on.

The retrieval commands are all processed in a similar manner by the logic of the retrieval mechanism. Figure 3.6.1 shows the flow of the retrieval logic; the following discussion should make it clear.

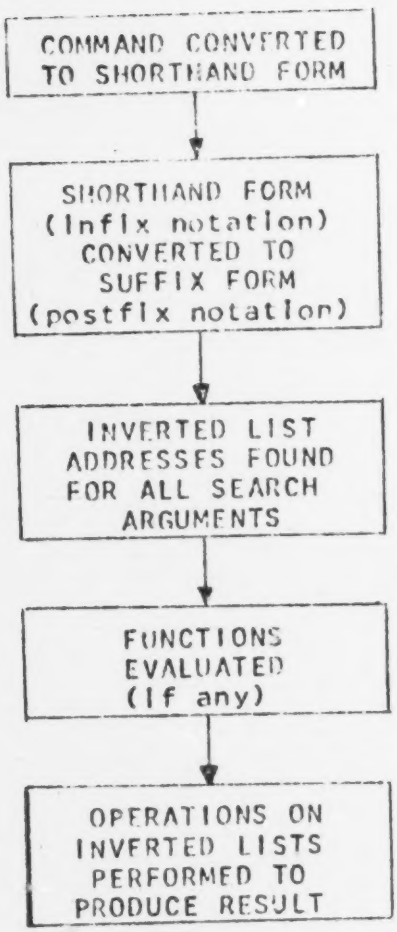

COMMAND CONVERTED TO SHORTHAND FORM
SHORTHAND FORM (Infix notation) CONVERTED TO SUFFIX FORM (postfix notation)
INVERTED LIST ADDRESSES FOUND FOR ALL SEARCH ARGUMENTS
FUNCTIONS EVALUATED (If any)
OPERATIONS ON INVERTED LISTS PERFORMED TO PRODUCE RESULT

Figure 3.6.1

Retrieval Processing

It will prove most instructive to follow the processing of an example through the logic. First, we will assume that our data base has several files, one of which is the following:

```
001 PERSONNEL FILE
    002 NAME
        003 FIRST NAME     (ITC=1)
        003 MIDDLE NAME    (ITC=2)
        003 LAST NAME      (ITC=3)
    002 AGE                (ITC=4)
    002 OCCUPATION         (ITC=5)
```

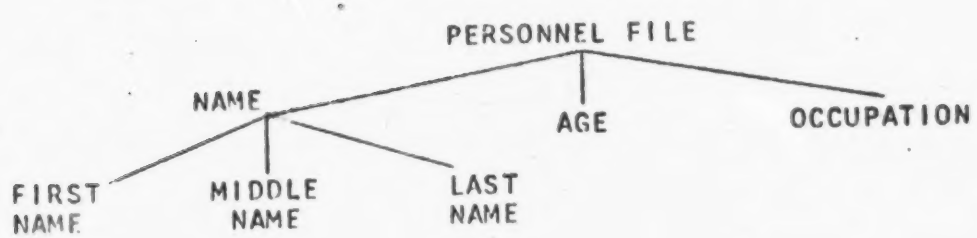


Figure 3.6.2

Sample File Structure

Let us take as our example the following command:

```
RETRIEVE NAME = SMITH OR (TOM AND JONES)
        AND AGE = BETWEEN(20,30)
        AND OCCUPATION = CAB DRIVER
```

Simply stated, the command asks for the records of every cab driver between 20 and 30 years of age who has either the word SMITH as any part of his name or both of the words TOM and JONES as any part of his name.

In its initial form, the command is awkward because of the extreme flexibility of possible constructions. To make the processing of the command more efficient, a translator from external to internal (template) form, the SHORTEN subroutine, was designed.

CODE	UNUSED	RELEVANT INFORMATION
1 byte	1 byte	1 halfword

Figure 3.6.3

An Element of The
Shorthand-Retrieve-Array

In the Shorthand-Retrieve-Array, SRA, each element of the retrieval command is assigned four bytes, one of which is one of the following codes:

'N' for category name;
'V' for search value;
'1' for level-1 operator;
'2' for level-2 operator;
'F' for function;
'(' for left parenthesis;
')' for right parenthesis;
'=' for equal sign.

Along with each code, a binary halfword contains the relevent information described below. The other byte is currently unused.

Since there is always a table of operators in core, each operator in the command can be identified by the pointer to its location in the table.

Because of this fact, whenever a level-1 or level-2 operator is found in the command, the appropriate code and pointer into the table is entered into the SRA.

In order to minimize manipulation of variable-length elements, all category names and search values, also called keys, appearing in the command are stored in the String-Array. Each element of the String-Array contains the length in halfwords (including itself) and the appropriate key. In order to preserve halfword boundaries (as in the directories), a blank is added to the end of the key if necessary.

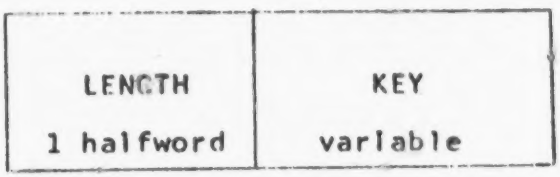


Figure 3.6.4

An Element of the String-Array

Whenever a category name or search value is found, the appropriate code and pointer to its location in the String-Array is entered into the SRA.

The binary halfword associated with parentheses and equal signs is disregarded. Because there is a table of functions, the decoding of retrieval functions is handled in the same manner as the decoding of operators. In addition, a special code of 'E' is used to denote the end of the arguments of a function; its binary halfword is ignored. The need for this special code will be realized later.

The decoding of the command and the formation of the Shorthand-Retrieve-Array are based on the following observations:

(1) The set of characters found to the left of the first equal sign is the first category name;

(2) The set of characters found to the right of the last equal sign can only be functions, search values, or level-2 operators;

(3) The set of characters found between two equal signs must be composed of those elements described above in (2) followed by one level-1 operator followed by a category name;

(4) Parentheses can occur anywhere, regardless of above observations.

The process starts by entering the first category name into the String-Array and putting the appropriate elements into the SRA. The equal sign is then entered into the SRA. The command is then scanned to find the next equal sign. If there is an equal sign, then there must be a category name immediately to its left, and a level-1 operator to the left of the category name. The processor uses this fact to decode the characters between the equal signs. The command is scanned backward from the second equal sign until a level-1 operator is found (by comparing command elements to the operator table). The characters between the operator and the second equal sign form the category name. Finally, the set of characters between the first equal sign and the level-1 operator must be composed of functions, search values, and level-2 operators. By

checking each element in this range against the function and operator tables, the correct code for each element can be determined. If a second equal sign was not found, the same decoding applies without the need for a level-1 operator and a category name. By using this process, the entire command can be easily decoded.

Special characters (i.e., other than numbers and letters) are treated separately. It is assumed that elements of a command are separated by spaces. In addition, special characters are taken as elements themselves. Hence, the search value MILES/HOUR would actually be decoded as three search values: "MILES" followed by "/" followed by "HOUR". In order to avoid this decoding, the intended value should be enclosed in single quotes, meaning "take the value literally". Hence, the values 'MILES/HOUR' would be decoded as desired. The use of quotes also allows the searcher to specify that a space is contained in a search value or that he is searching for a word which is the same as a function or operator name (e.g., searching for the word "AND"). The special character handling allows easier decoding of single operator symbols, such as "+" or "&".

The decoding process is designed to recognize only the most severe syntactical errors, such as a retrieval command without an equal sign. The major part of the format error processing is done in the next stage of processing.

It is assumed, here, that "AND" is the first operator, "OR" is the third operator in the operator table, and "BETWEEN" is the second function in the function table. Also, the pointers for N's and V's point to the length halfwords of the appropriate String-Array elements.

Now, if we follow our example through the decoding, we can understand the formation of the following tables:

N		1
=		
V		4
2		3
(		
V		8
2		1
V		11
)		
1		1
N		15
=		
F		2
V		18
V		20
E		
1		1
N		22
=		
V		28
V		31

Figure 3.6.5

SRA Example

03	NAME
04	SMITHB
03	TOMB
04	JONESB
03	AGEB
02	20
02	30
06	OCCUPATION
03	CABB
04	DRIVER

Figure 3.6.6

String-Array Example

With the command in the internal shorthand form, the next step is to translate the infix form to a postfix (suffix) form. In addition, the category names can be converted to the appropriate ITC's. Finally, syntactical error conditions are intercepted at this time. All of this processing is performed by the TOSUFFIX subroutine.

The elements of the SRA are processed sequentially, with detected errors displayed to the searcher as they are encountered. In addition,

certain assumptions are made to resolve each syntactical error condition. After syntax checking is completed, if at least one error is found, the template of the assumed command is displayed and the searcher is asked whether the assumptions are satisfactory. If they are, processing continues; if not, the command is aborted.

The output of the TOSUFFIX processing is the Suffix-Array, which is a set of six-byte elements. These elements, similar to SRA elements, contain a one-byte code and two relevant halfwords. The Suffic-Array codes are:

'V' for search value;
'F' for function;
'E' for end of function;
'1' for level-1 operator;
'2' for level-2 operator;
'G' for generated operator;
'J' for jump element;
'A' for inverted list address;
'*' for end of array.

For search values, the first halfword is still the String-Array location; the second halfword is the ITC to be searched for. Notice the actual search value is not accessed from the String-Array at this point.

The jump element will be explained in a later section; the end of array element is simply the last element to be processed.

A template of the command, to be displayed to the searcher, is formed from the SRA as it is translated. This template uses exactly the SRA codes. During the course of inverted list processing, if the

searcher desires it, a trace of the logical operations is displayed; that is, the number of records resulting from each operation is printed out directly under the appropriate code in the template. It is therefore necessary to associate an element of the template to each operator. For each of the three types of operators, the second halfword in the Suffix-Array is the pointer to the template. In addition, since a function generates an "OR" operator between each resulting value, the function element in the Suffix-Array is treated as a generated operator.

When a category name is found in the SRA, a set of subroutines is called to convert a name to one or more corresponding ITC's within the current qualification. This sets up the searching of the search value with each ITC appropriate to the given name. If more than one ITC is associated with a category name, then an "OR" operator is generated to connect the sets of search values for the multiple ITC's.

As the SRA is sequentially read, each element is acted upon immediately. As stated above, a category name is converted to a set of ITC's. Until a level-1 operator code or the end of the command is found, all search values entered into the Suffix-Array have the first ITC of the indicated set associated with them. When the level-1 operator or end of command is found, all entries put in the Suffix-Array since the category name was found are duplicated with the second ITC in the set (if there is one) replacing the first ITC. Then, the generated OR operator is entered to operate on these sets of entries. This duplication continues until all ITC's for the category name have been exhausted. For instance, if a command were RETRIEVE NAME = A, then the Suffix-Array would have an entry for A with the ITC for FIRST NAME, an entry for A with the ITC for MIDDLE NAME, a

generated OR operator, an entry for A with the ITC for LAST NAME, and finally another generated OR operator. In effect, the command asks for the data item A occurring in either the FIRST NAME, MIDDLE NAME, or LAST NAME category. This suffix form of the command represents precisely the same query.

When a search value code is found, it is immediately entered into the Suffix-Array with the first ITC in the appropriate set as described above. When a function is found, the group consisting of the function, its search values, and end of function code is treated as a single search value and entered into the Suffix-Array with the appropriate ITC.

The processing of level-1 and level-2 operators is more intricate. First, it should be mentioned that each operator in the operator table has precedence associated with it (i.e., some operators are to be evaluated before others). As an algebraic analogy, consider the equation X = 3*4+1. If the addition is performed first, then X is 15; if the multiplication is performed first, then X is 13. Normally, the multiplication is the first operation to take place because it has "higher precedence". However, if the equation were X = 3*(4+1), the addition would take precedence because of the parentheses. One can see the flexibility that this type of precedence structure allows; this is why the logical operators of the retrieval requests are given precedence.

To demonstrate the translation to suffix using precedences, assume a command requests information for A AND B OR C is some category. The code for A is placed in the Suffix-Array. The code for AND is put in an operator stack. Then, the code for B is placed in the Suffix-Array. Now, the precedence of OR is compared to the precedence of AND. In the

case where the precedences are equal, either of the cases below is possible depending on the implementation of the algorithm, but both interpretations are legitimate. There are two cases to consider:

Case 1. AND has higher precedence than OR. In this case, the code for AND is removed from the operator stack and placed in the Suffix-Array because it has higher precedence. Then the code for OR is put in the operator stack and the code for C is placed in the Suffix-Array. Now that command elements are exhausted, the operator stack is emptied into the Suffix-Array. Hence, in suffix form, the command appears to be A B AND C OR; this means "find records with either C or both A and B".

Case 2. OR has higher precedence than AND. In this case, the code for OR is put in the operator stack after the code for AND. Then, the code for C is placed in the Suffix-Array. With the command elements exhausted, the operator stack is emptied in reverse order (last in, first out) into the Suffix-Array. Hence, in suffix form, the command appears to be A B C OR AND; this means "find records with A and either B or C".

Parentheses are handled in a simple manner. When a left parenthesis is found, it is entered into the operator stack as if it were an operator with the lowest precedence. Then, the processing continues with the left parenthesis remaining in the stack because of the algorithm. When the

matching right parenthesis is found, all stack elements after the left parenthesis are emptied in reverse order into the Suffix-Array. The parentheses are discarded, having served the purpose of defeating precedences.

When the problem of precedences is further complicated by two levels of operators, the solution is simple. Whenever a new level-1 operator is encountered, its precedence is compared to the previous level-1 operator in the stack. If the new one has higher precedence, it is added to the stack. If the new one has lower precedence, then the previous level-1 and all intermediate level-2 operators are emptied in reverse order into the Suffix-Array. Then, the new one is added to the stack. At the end of the command, whatever operators are left in the stack are emptied in reverse order into the Suffix-Array.

It is important to note that the searcher is allowed by the SET command to change the precedence of any operator at any time. This is a powerful tool if used properly.

The Suffix-Array for the original example is shown in Figure 3.6.7. It is assumed that OR has higher precedence than AND, but note the placement of the parentheses. The template of the command is:

N=V OR (V AND V) AND N=F(V,V) AND N=V V

The second pointer for operators in the Suffix-Array is the position, counting all characters and spaces, of the element in the template. Generated operators are associated with the equal signs in the template. Later, the operators generated by the function are associated with the F in the template. Note that the space between the two V's at the end of

V		4	1
V		8	1
V		11	1
2		1	12
2		3	5
V		4	2
V		8	2
V		11	2
2		1	12
2		3	5
G		3	2
V		4	3
V		8	3
V		11	3
2		1	12
2		3	5
G		3	2
F		2	24
V		18	4
V		20	4
F			
1		1	19
V		28	5
V		31	5
2		5	38
1		1	32
*			

Figure 3.6.7

Suffix-Array Example

the template is now a level-2 operator. It is the implied concatenation operator (assumed to be the fifth operator in the table).

Up to this point, the directories have not been accessed. But, in order to perform the operations on the inverted lists, the list addresses must be found. The DIRSRCH subroutine converts each search value in the

Suffix-Array, except arguments of functions, to the appropriate inverted list address.

The procedure used to search for a key is the following. A binary search of the Core Directory isolates a single directory at the next level. It is read from disc; if it is a high-level directory, the binary search is applied again and the process repeats. When a low-level directory is finally read, it is guaranteed that if the key exists, it is here. A linear search of the low-level directory determines whether the key exists or not. If it does exist, the associated ITC's are searched also.

In the searching for keys from the Suffix-Array, the values must be extracted from the String-Array. Even though the same value occurs with different ITC's in the Suffix-Array, it is searched only once; all of the ITC's are evaluated at the same time. After the search is performed, the entry in the Suffix-Array (V, String-Array pointer, ITC) is replaced by a code of 'A' and the system address of the inverted list; the address occupies the two halfword pointers. If the value or its ITC is not found by the search processing, a special address (all binary zero) is placed in the Suffix-Array.

In the case of our example, the only values left in the Suffix-Array after directory searching are "20" and "30" which are the arguments of a function. All other values have been converted to addresses.

Before the operations on the inverted lists can be executed, the functions must be evaluated and converted to inverted list addresses. The TOADDRE subroutine handles the functions.

A problem is caused by evaluation of functions. The execution of operations on the inverted lists is based on a sequential reading of the Suffix-Array. However, the number of resulting list addresses may be far more than the reserved space for the function (for BETWEEN, four elements: F V V E). With this in mind, the jump element (mentioned previously was designed. The jump code is an indication to the list processor to branch to another location in the Suffix-Array and resume sequential execution there. The first halfword associated with a jump code is the location of the next element to process in the Suffix-Array; the second halfword is unused.

For each function, the F in the Suffix-Array is replaced by a J; the jump pointer is set to the location of the next empty location in the array. The results of the function are entered into the array starting at this location. At the end of the evaluation, a jump code and jump pointer are entered to transfer execution back to the element after the E associated with the function.

The results of a function are logically connected by generated OR operators (e.g., BETWEEN(20,30) means 20 or 21 or 22, etc.). Of course, the operators and addresses are placed in the Suffix-Array in suffix form.

The functions are evaluated by first using the directory searching routines to find the key specified by the function. Then, the appropriate low-level directory (and possibly adjoining low-level directories) are scanned linearly to find other keys in the functional range. The searcher is allowed to specify a function limit; that is, function evaluation for any single function ceases when the number of addresses found is equal to the limit.

Assuming that three ages are found between 20 and 30, then the Suffix-Array of the example is shown in Figure 3.6.8.

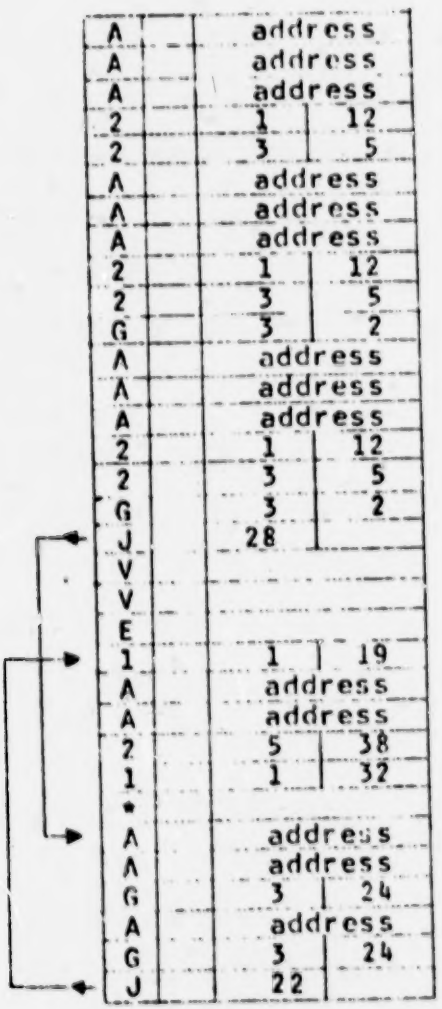


Figure 3.6.8

Suffix-Array Example

The final step in the retrieval process is the execution of logical operations on the inverted lists (PHASEIV subroutine). As mentioned

before, the elements of the Suffix-Array are operated on sequentially (with jump elements redirecting the sequence). The only codes allowed at this stage of processing are addresses (A), operators (1, 2, G), jumps (J), and the end of the array (*). The processing of operators is independent of their type (level-1, level-2, or generated). They are differentiated simply for the trace facility; the searcher can request tracing of any one or all of the three types of operators.

Because an operator applies to the previous two operands in suffix form, the addresses must be stacked up in an operand stack. As the Suffix-Array is read, each address is put in the stack. When an operator is encountered, the two addresses most recently added to the stack are pulled out. The operation on these two lists produces an output list; the address of this output list is then added to the stack. When the end of array code is finally found, the operand stack should contain only one address. This is the address of the active list which the command has produced.

There are three 2000-byte areas in core for use by the operator routines. Any two of these can be used for the input lists (inverted lists to be operated on); the third is used for the output list. Before the operation is performed, the two input lists are read into these areas (if not already there from a previous operation), and a cylinder in the work file is allocated for the output list about to be created; only 2000 bytes of any list, input or output, can reside in core at any time. The cylinder allocation allows for shorter access time when manipulating segments of the entire list; also, the output of one operation will probably be the input of a later operation in the array. When an operation is completed, the two input list addresses

are checked; if either is in the work file (the output of a previous operation), its cylinder is deallocated because it is no longer needed. In this way, the work space is re-used efficiently.

For each operator in the operator table, there is the name of the subroutine used to perform the operation. These subroutines are loaded dynamically; that is, no operator routine is loaded into core until it is called the first time. The routines are supplied the first segment (physical record) of each input list, and the area for the output list. In addition, a count of records in the output list is produced for tracing purposes.

There are five operators in the current implementation of SOLER. The first three are both level-1 and level-2; the last two are only level-2.

1. AND - produces an output inverted list containing one entry for each IRN which occurs in both input lists; the user bytes are not checked.
2. OR - produces an output inverted list containing one entry for each IRN which occurs in either input list; the user bytes are not checked.
3. ANDNOT - produces an output inverted list containing one entry for each IRN which occurs in the first input list but not in the second input list; the user bytes are not checked.
4. CONCAT - produces an output inverted list containing every entry from the second input list whose IRN and first user byte (sequence number) are identical to and

second user byte (word number in sentence) is one greater than the corresponding elements of an entry in the first input list.

5. SEQUENCE - produces an output inverted list containing every entry from the second input list whose IRN and first user byte are identical to and second user byte is any amount greater than the corresponding elements of an entry in the first input list.

In the example, the AND and OR operators need no explanation. The CONCAT operator (implied by the space between CAB and DRIVER) is interesting however. If an AND operator has been specified instead of the concatenation, the retrieval processing would have found records for "DRIVER OF CAB REPAIR TRUCK". This is obviously not the same occupation. In searching the directories, let us assume a list for CAB and a list for DRIVER was found for the proper ITC. When the CONCAT operation is performed, the sentence and word numbers (user bytes) are checked. In "DRIVER OF CAB REPAIR TRUCK", the word DRIVER does not directly follow CAB; hence, this record would not survive the operation. This phase retrieval capability is clearly an advantage in text-oriented data bases.

At the conclusion of the list processing, one inverted list remains in the work files as the result. If a restriction is in effect, an ADD operation is applied between the resulting list and the restriction list. The output of this operation is the new active list. The old active list is deallocated, the new active list is read into core, and the retrieval processing is completed. The searcher can then print from the records in

the active list; or, if he is not satisfied with his results, he can issue another retrieval command.

The APPLY command is processed in the same manner as has been described. However, before the old active list is deallocated, a level-1 operator is applied between the old and newly-created active lists. The result of the application of this last operation is the new active list. In other words, the APPLY command allows new conditions to be added to the records in the active list.

The REPEAT command simply re-executes the list processing of the elements in the current Suffix-Array. This is useful for changing the trace and re-retrieving.

The retrieval mechanism which has been described is so powerful that the casual searcher may never use some of its features. Use of the trace and precedence operators, for example, allows the searcher to issue a complicated request and still see his intermediate results.

One of the problems of this implementation is the fact that each searcher may have his own copy of the retrieval programs (about 250 thousand bytes) in core. To convert the system from one-user to multi-user, the programs should be recoded in a re-entrant manner. Such an implementation, however, would tend to tie the system to a particular machine.

Because of the design of the input phase of SOLER, external data may be encoded when entering the system. For instance, if data entered in the age category were specified in units of months, it could be stored, internally, in units of years; the input phase is designed to support such translations. The retrieval mechanism could (but currently does not) support decoding (the inverse of the encoding process performed

in the input phase). This decoding process would be invoked prior to directory searching to insure search of the proper keys.

3.7 The Output Phase

The Output Phase allows the user to display data from the records selected by the Retrieve phase. The user may select one or more data items, specify the order and format in which the data should be displayed and choose the destination of the data. The data may be displayed on the users terminal, printed on the high-speed printer, or transcribed to a file on disc for later processing.

The records to be processed by the output phase are those records currently on the "active list" generated by the Retrieve phase. This list is in ascending order by record number, and contains a pointer to the next list element (record) to be processed. This pointer is initially set to the first element, but will be modified by the processing in the Output phase. The user may also modify this pointer by using the special list manipulation commands.

Commands and Parameters

There are three commands in the Output phase: PRINT, LIST, and CONTINUE. The PRINT and LIST commands direct the system to output data. The CONTINUE command simply continues displaying the data specified by the most recent PRINT or LIST command.

In addition to these commands, there are several parameters which affect the format and destination of the output. These parameters may be changed by using the SET command.

This section discusses in detail the commands, parameters, and internal structure of the Output phase. These commands are also discussed in the User's Manual contained in Appendix A.

The PRINT Command

Format

PRINT (N,) <NULL>

NONE

<FIELD NAME>(,<FIELD NAME>(,...))

The N, if present, specifies how many records are to be processed. If N is omitted, all records in the active list will be processed. If the end of the active list is encountered before N records have been processed, the command halts normally and resets the pointer to the first record in the list.

If the operand field is null, all records will be printed in full. Data items will be printed in the order in which they appear in the record.

If the operand field contains the special keyword NONE, no data will be printed. Instead, the system will list the internal record numbers of the records on the active list.

The most common form of the command specifies the names of the fields which should be printed. These field names may be qualified, and are subject to any qualifications in effect from a previous QUALIFY command. The fields may be requested in any order. If a field is requested which does not appear in the record, the field name is skipped for that record and processing continues normally.

If a field name is specified which does not fall within the scope of the current qualification, or if an illegal field name is entered, an error message is printed and the command is terminated. The user should correct the error(s) and re-enter the command.

The destination of the output of the PRINT command is the user's terminal, unless overridden by the SET command.

The LIST Command

Format

< same as for PRINT >

The format and operation of the LIST command is the same as the PRINT command except that the destination of the output will be the high-speed printer unless overridden by a SET command.

The CONTINUE Command

Format

CONTINUE (N)

The CONTINUE command will continue the operation of the most recent PRINT or LIST command. All operations will be the same as if the PRINT or LIST command has been re-entered, except that the number of records to process, N, can be specified in the CONTINUE command. If N is not specified, all remaining records on the active list will be processed.

Structure of the Output Phase

The Output phase is interrelated with the Update phase. See the section on the Update phase for a discussion of the details which are applicable to the Update phase. This section will discuss the routines which control the sequence of processing to determine which records to process and which fields within the records should be examined.

Major Routines

The major routines in the Output phase, and the action performed by each routine, are listed below.

OUTPUTPH - This routine is the control routine for the Output and Update phases. There is an entry point in OUTPUTPH for each of the commands in the Output and Update phases. When the executive determines which command should be executed, the correct entry point in OUTPUTPH is called. Within OUTPUTPH, flags are set in the SYSTEM-DATA-AREA to indicate which command is being processed. The COMPILE entry point in PC[illegible]PILE is then called to process the operand field of the command. This sets up the field names in the PRINT-TABLE in a coded form, and returns the number of records that should be processed (if specified by the user). If the user did not specify an explicit number, an extremely high number is returned as an approximation of infinity.

Once everything is set up, the routine loops through the "active list" or record numbers to process the records. Each IRN is extracted from the list, and the entry point PRINTREC is called in the routine PRINTA.

PRINTA - This routine is used for a programming trick more than any logical need. The IRN of the record to be processed is passed in from OUTPUTPH. The IRN is converted to a file address by calling IBNCONV. The data record directory for the record is read into a common buffer that will be referenced by the routines which actually process the record. A check is made to see what definition should be used to interpret the record, and the definition is loaded into the main buffer. Now the PRINT entry point of PRINTB is called. Since the pointers are available which say where each piece of the definition is located in the main buffer, when PRINTB is called this can be passed as an absolute address. Thus within PRINTB, the tree array, ITC array, etc., can be defined on a logical level, ignoring the structure of the main buffer.

PRINTB - This routine is responsible for interpreting the definition, the data record directory, and the PRINT-TABLE to determine which data items in the record should be processed (either for printing or updating). The PRINT-TABLE contains in coded form the list of fields which the user requested. Each element in the table is a three halfword entity. The first halfword is the number of the definition which contains the fields specified by the user.

Since the user may specify fields from many different record types, not all elements in the table will apply to this record. Any element which does not apply is simply ignored. The second halfword is an index, referring to the tree array of the definition, which points to the element in the tree array which is the logical start of the field requested by the user. The final halfword is the index of the last element contained in the field requested by the user. Since the user may specify a non-terminal field name in a PRINT request, there may be many elements contained in the section of the tree array bounded by these pointers. For example, if the user specifies that the entire record should be processed, the first pointer would be a 1, and the second pointer would point to the last element in the tree array. The routine DRDSCAN is called to actually scan the data record directory to pick out the elements to be processed. When an element is found, DRDSCAN returns with a pointer to the element. PRINTB determines which field names (if any) should be printed, and does the appropriate I/O.

To actually process the data, the MINIEXEC is called. This routine is the start of the Update phase processing, and also serves to print the data for the Output phase.

DRDSCAN - This routine scans the Data Record directory of a record to pick out the next data element to be processed. When an element is found, a pointer is set and returned to the calling program. This routine is designed to handle subscripted field names, but the PCOMPILE routine is not yet able to compile subscripts into the internal form needed by DRDSCAN.

Figure 3.7.1 shows the interconnection of the routines in the Output phase.

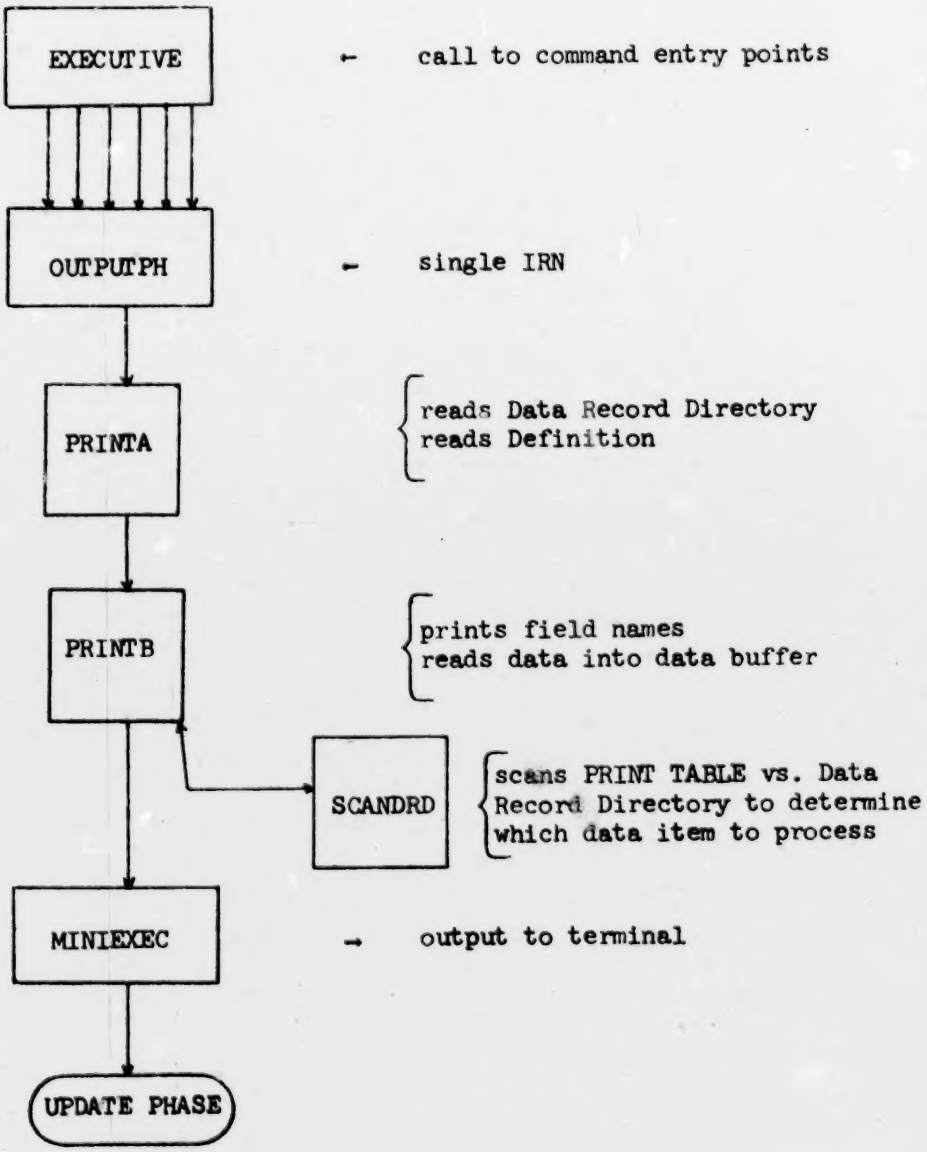


Figure 3.7.1

Macroflowchart of Output Phase

4. STATISTICAL CLUMPING AS AN INDEXING AID

4.1 Adaptive Interaction

In many document retrieval systems, there is one aspect of the operation that requires further development, namely the perfecting of communication between man and machine. Specifically, the user obtains little assistance from the machine or system when he is attempting to state his request. The user is confronted with a word list of all available index terms from which he must make a selection to enter his search request. A study was conducted on this project to gain insight into the problem of organizing the information that can be placed into the system in the form of index terms and to use this information to render assistance in the formulation of a request.

The study demonstrated a process whereby the user and the system adapt to each other's need and viewpoint. This adaptation on the part of the user takes the form of his accepting the system vocabulary terms and their interrelations. The adaptation on the part of the system allows it to modify the manner in which it proposes new index terms and to give the user advice that guides him in the selection of his index terms.

The study provided three specific contributions to the field of information retrieval:

1) A framework of man-machine adaptive interactive conversation providing unsolicited librarian-like assistance to the searcher through increasingly better sets of index terms. The data set for the interaction is based on the second contribution.
2) A technique to extract pairs of related terms from a set of machine-readable English documents. The technique eliminates common terms, distinguishes general conceptual terms from particular terms and extracts pairs of terms that exhibit conceptual relationships with the particular terms through frequent association in selected text. The particular terms are essentially the equivalent of descriptors in an index of documents.
3) An algorithm to assist in the development of a thesaurus. The algorithm ascertains sets of nearly mutually synonymous terms when given a table of synonyms in machine readable form.

The adaptive man-machine interaction contribution provides a technique for the information retrieval system to render assistance in the formulation of index sets without the burdensome task of the user reading abstracts of recovered documents. Without this assistance, the task of refining an index set usually rests upon the retrieval of abstracts and titles of a set of documents and their evaluation by a user. This ad hoc index set production results in an uneconomical searching of mass storage before the topic is clearly defined by a suitable index set.

The system presented that was developed provides the necessary assistance to allow the refinement of an index set before searching document files. The data base employed by the system is a set of relation tables produced semiautomatically from a collection of documents in machine readable form. These relation tables are two dimensional, weighted, topic differentiated, and are produced in a semiautomatic manner. They employ the contiguity relation; i.e., terms that appear within a given distance of each other frequently are highly related. Given a preselected file of documents, terms that appear often within a given distance from each other render a high contiguity measure. In order to be fully effective, the raw samples of English text from which the data base is derived must be representative of the types of documents that the information system is called upon to recover. Pragmatically, many of the actual documents in the document file are used in the derivation of the relation tables. The technique for generating the relation tables from the English strings will, with little human intervention, produce sets of relation tables that are topic differentiated. In the following, these tables shall be named "microthesauri" meaning a thesaurus, or relation table, of a small limited topic area.

The algorithm that constitutes a central tool for the adaption process ascertains the sets of nearly synonymous terms from a table of synonyms. The concept of near synonyms is treated at length in a formal manner later; but, for the present purpose, two terms may be considered nearly synonymous if there exists at

least two completely separate sequences of synonyms connecting them.* The use of this technique eases the production of a table of synonyms by checking for sets of nearly mutually synonymous terms as the table is generated. The technique has application in any process that uses an algorithm to produce word pairs rich in synonyms. As these synonymous pairs are included in the table of synonyms, the algorithm assists the user to perform consistency and completeness checks.

The process developed in this experiment exhibits two forms of adaptive behavior. The first form is the ability of the system to respond to an index set proposal in a manner that depends upon the previous history of the man-machine dialogue. The system does this by preparing a profile of interest during the interaction and uses this interest profile to frame its replies. The profile consists of numbers or scores that are related to the level of interest that the user has developed in each of a set of microthesauri. The second form of adaptive behavior is the ability of the system to render suggestions and to direct the interaction in accordance with its estimate of the user interest profile.

The system can determine at least three types of directions in any interaction between man and machine:

---

* For example: Facile-Easy-Smooth and Facile-Simple-Smooth would imply that Facile and Smooth are nearly synonymous.

1) A tendency for the profile of interest to centralize on a small group of microthesauri--in this case, the user is probably refining his search into a small set of areas and should be apprised that he is reaching a terminal stage in his interaction.

2) No apparent trend in the pattern of the search--perhaps the user is browsing or he does not understand the function of the system.

3) A tendency toward an "oscillation" in the interest profile--perhaps the user has been diverted in his search, or he has changed emphasis intentionally.

The action taken by the system ranges from the advisory guidance message to pointed suggestions, and ultimately to the suggestion that human intervention may be required.

4.2 Implementation of the System

The system was implemented on the IBM 7040-PDP 8 computer complex with remote Teletypewriter input. The 7040 portion of the implementation was programmed in L6 linked list programming language. All the functions needed to implement the interaction were included. The adaptive algorithm was only partially implemented; full implementation required only the gradual accumulation of relation tables for a complete data base.

4.3 General Definitions

In this section, a number of terms will be introduced that are employed in the body of the work. These terms will be underlined the first time they are introduced.

A word type is any string of characters except blanks and terminal characters that expresses a particular meaning for the user of the system. Also, the system can have instances of words in its internal vocabulary. A term is a word (or perhaps a word phrase) that the system has in its vocabulary and, in addition, the system employs it in one of a number of different applications. In particular, the system differentiates among the following types of terms. A common term is a word that is considered by the system to be used so frequently in all fields of natural English as to render it useless when a particular subject is to be recognized. Examples of common terms are "description", "although", and "specifically". A core term is not a common term but appears in one particular subject area so frequently that it is associated with the area and, in fact, is useful in identifying it. However, the core term is not useful when called upon to discriminate among documents in its field. A core term is useful as a way to raise the recall level of an index set. Examples of core terms from the field of computing software might be "computer", "compiler", and "system". A particular term is a term that appears in the document set of a field of interest in a manner that it is useful in discriminating among the documents of the subject area. A homograph is a word or term that has two or more independent meanings. It is possible for a term that has a homographic nature to have at once any combination of common, core, or particular meanings.

The notion of a descriptor is well established. It is a

term that is used to give a clue concerning the subject content of a particular set of documents. An index set is a set of words the user employs to characterize a particular subject in which he has an interest. The function of the interactive process is to transform the index set of words into an index set of descriptors that are rich in particular terms to give a crisp definition of the request. An adaptive process is a technique whereby the system is able to respond in a manner that depends upon both the past interactions and the present state of the user's index set.

A thesaurus is a set of relations whose left and right components are terms in the system vocabulary. Formally:

$$T = \{ R \mid R \epsilon \text{Relation} \ \& [(At) (t \text{ is a left or a right component of } R \Rightarrow t \epsilon T \text{ where } T \text{ is the system vocabulary})] \} \qquad 1\text{-}1$$

In the current stage of implementation there is only one relation; it is a contiguity relation derived semiautomatically from a textual data base.

A microthesaurus is a portion of a thesaurus for which the following applies:

$$M_a \epsilon T \ \& [(At)(t \text{ is a right component of } R \Rightarrow t \epsilon T_a \text{ where } T_a \text{ is a set of particular terms in subject area "a"})] \qquad 1\text{-}2$$

Thus, a microthesaurus is a portion of a thesaurus for which all the right members of the ordered pairs defined by the microthesaurus are restricted to be particular terms in the subject area of the microthesaurus.

A specialty area is a portion of a field of interest that may be identified as having sufficient internal cohesion in its subject matter to be discovered both statistically and subjectively. An interest profile is an a-tuple of ordered pairs as follows:

$$P_a = (G_a(N), L_a(N)) \qquad 1\text{-}3$$

Where: the a'th pair of ordered terms is associated with the a'th subject area,

And: $G_a(N)$ is a measure of the degree of interest expressed by the user in the a'th area after N interactions, expressed as a microthesaurus gain,

And: $L_a(N)$ is a measure of the level of detail or degree of specificity expressed by the user after N interactions.

4.4 Theoretical Considerations

This experiment was based on the foundation principle that the field of document retrieval requires at least two further developments before it may be accepted as a working tool. These are the development of a mode of truly interactive "understanding" between the user and the system and the production of

better methods of extracting clues that characterize the document from the documents themselves; i.e., index them. These two aspects of the document retrieval field will be reviewed in the following introductory remarks.

In the course of history, man has gradually developed a variety of languages for communicating and recording an ever increasing variety of complex ideas. These languages have grown with almost no control or formal justification. In fact, some believe that it is this lack of control that provides for the richness of natural languages and their great capability to provide a varied means of communication among individuals that speak the same dialect of their language.

However, this same absence of analysis or understanding of the formal implications of the language has impeded the development of facile communication between man and his machines. As long as the machines that man built were of the purely slavish responsive type, this communication problem was of little consequence. In many cases, the only necessary means for communication was the provision of a set of knobs and levers attached to the machine. When the large scale digital computer was realized, man conjectured that the large memory and the "giant brain" of the computer would give him the ideal machine with which he could converse on more than a trivial level.

Workers in the computer field began the complicated analysis of natural language. As time passes, these attempts at analysis seem to uncover successively more complex problems that have definitely impeded efforts to effect natural

communication between man and his machines. Until there exists a mechanism that allows the machine to "understand" man's written or spoken words, some less ambitious technique will have to be provided for such communication.

The present work considers a particular conversational mode that is therefore less than fully interactive "understanding" conversation. What is provided is a framework of a conversational mode with enough informational content to enable a machine to respond "intelligently" in a man-machine interaction. The function of this interaction is not to "explain" anything to the machine, but to transform the information presented to the machine by the man into a form that is both meaningful to the machine with respect to its own stored intelligence, and also preserves or inferentially improves the intent of the original statement of the man.

This transformation is made after the machine receives words from the man and then changes or reformulates them within the framework of the stored information that the machine has at its disposal. This process of man-machine symbiosis is referred to as an adaptive interaction in the following sections of this work. The interaction is a two-way adaptive process because both man and the machine adapt to each other. The adaptation of man is to information [illegible]s structure in the machine; the adaptation of the machine is to [illegible] background and current interest viewpoint of the man.

Suppose that a user's index set is represented by the letter U and that the i^{th} such set would be U_i. Let the machine response be M_i. In general, there will be a sequence of requests and responses in an interaction. If the interaction is I, then:

$$I = U_1M_1U_2M_2 \ldots U_nM_n \qquad 4\text{-}1$$

At any point of the interaction, the response is a function of all that has preceded it:

$$M_i = f(U_1M_1U_2M_2 \ldots U_i) \qquad 4\text{-}2$$

This can be re-written in terms of the present input and the last response:

$$M_i = g(U_i) + h(M_{i-1}) \qquad M_o = \text{null} \qquad 4\text{-}3$$

The system that is represented by Eq. 4-3 is similar to the familiar feedback equation. The scenario that will be developed is one in which the user supplies the first index set, and the machine after consulting the data set and some internal memory, responds with a set of its suggested terms. The user then is allowed to evaluate these terms and by the evaluation, the system ammends its internal memory. The user's reaction initiates the second and succeeding interactions. In order to set the stage for the development of an adaptive system, it is necessary to consider the implications of the term adaptive.

4.5 Preliminary Definition of an Adaptive System

There is no general agreement on the formal definition of what is meant by an adaptive control system; it is generally accepted that an adaptive control system is one that is capable of noting the environmental conditions that prevail and modifying the system performance on the basis of the changes it perceives. The standard notion of an adaptive control system has been defined by Eveleigh and is quoted below:

> "An adaptive (control) system is one which is provided with a means of continuously monitoring its own performance in relation to a given figure of merit or optimum condition and a means of modifying its own parameters by closed loop action so as to approach this optimum."

After introducing this definition, the author cited three factors that are essential to an adaptive system:

(1) Identification

(2) Decision

(3) Modification

Identification determines the factors that are needed to characterize optimum operation of the system. Decision determines that a change is needed in the system operation in order to achieve the optimum operation. Modification determines how to change the system parameters to utilize the results of the decision. Even with these definitions and comments there still is no general agreement on the definition of an adaptive system.

The above discussion is based primarily on analogue or hybrid systems and is not immediately applicable to the case of a digital computer environment, the reason being that the computer is, by its nature, able to modify its own process (program) in order to achieve better system performance. Even such a simple digital computer as the telephone central office is able to modify its mode of operation when it decides that the environment is such that the system parameters should be changed to achieve the optimum operation that has been identified as the goal of the system. An example of this is the "automatic line load control facility that allows the telephone system to accept only emergency calls under certain conditions. In the case of complex digital computers, the adaptive mode is present but obscured by the fluctuations in the programs caused by outside effects.

4.6 Habit Forming vs. Learning

The present selection will consider an example of a system where the modification is explicit and treated formally. Various forms of a mechanical processor that exhibit habit-forming and learning are presented by Gorn. He defines habit-forming and learning as follows:

> "Habit-forming: A process that involves a selection of one of many alternatives where the selection is more likely to be made because it was selected at some previous time.
>
> "Learning: A process that envolves a selection of one of many alternatives where the selection

is more likely to be made on the basis of the results of previous selections." [22]

The difference between habit-forming and learning is that in the former, the habit is reinforced merely because it has been chosen. In the latter, learning occurs as a result of an evaluation influenced by external results. People exhibit both habit-forming and learning; an example from the experience of the author is presented:

If a beginning programmer writes a program in a certain way; e.g., without the use of any comments in the code, then he will probably continue to write programs that way (regardless of the comments of his co-workers). He has acquired a habit. Now if our tyro programmer is ever called upon to rework one of his pieces of code after time has elapsed, it is a good bet that he will learn the advantage of annotated code. If our programmer is intelligent, he will learn to mend his ways. Then it can be said that learning was acquired in an environment.

In order to appreciate the development of the habit-forming processor, consider Fig. 4-2. After Fig. 4-2 is developed, it will be shown how the present system is similar.

In Fig. 4-2, each input is derived from the last output. The initial output is set by some external source. At each cycle, a random number is generated and compared with a set of probabilities, $\bar{\rho}$, where $\bar{\rho}$ is given below:

$$\bar{\rho} = \{\rho_1, \rho_2, \ldots, \rho_n\}$$

$$\rho_i \geq 0 \quad \text{for} \quad i = 1,\ldots,n \quad \text{and} \quad \sum_{i=1}^{n} \rho_i = 1 \qquad 4\text{-}4$$

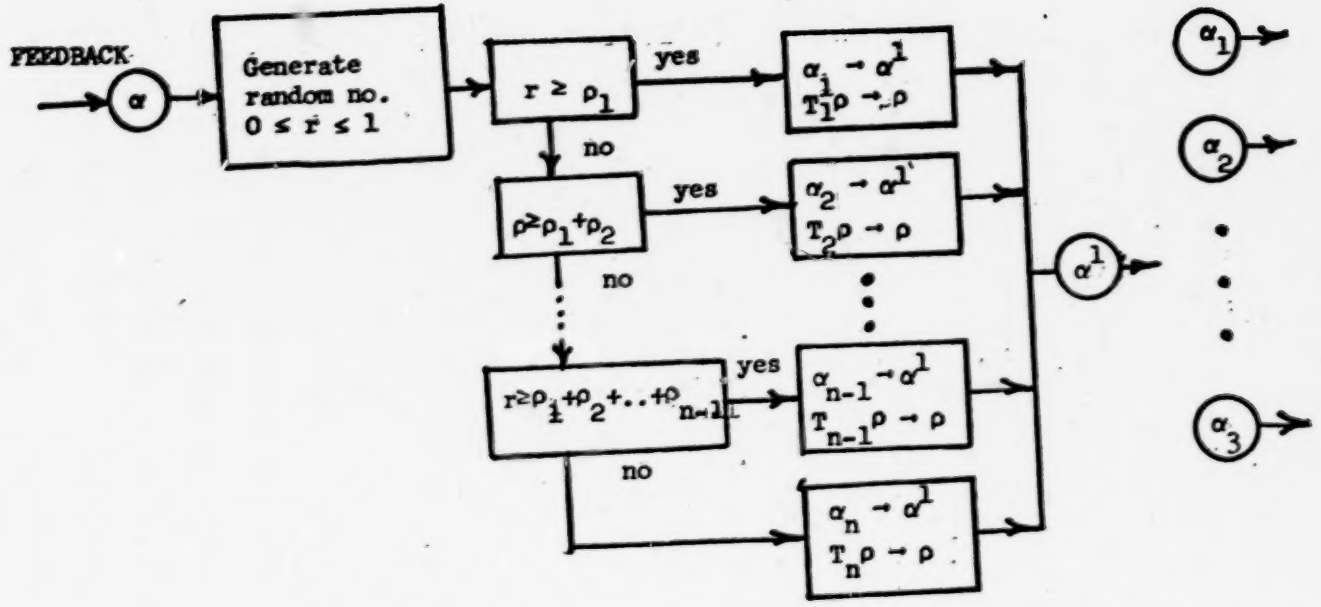


Fig. 4-2 - General Habit-forming

Suppose that there is a set of operators T_i, and that when T_i operates on $\bar{\rho}$, the resultant value of $\bar{\rho}$ is changed so that the values of ρ_i are increased at the expense of the remaining components of $\bar{\rho}$. In operation, each cycle yields a random number r, the value of which is used to select the next value of the output α^1. At this time, the associated T operator modifies the vector $\bar{\rho}$. As an example of this, suppose that:

$$T_1\bar{\rho} = T_2\bar{\rho} = \ldots = T_n\bar{\rho} = (\rho_2,\rho_3,\ldots,\rho_1,\rho_n) \qquad 4\text{-}5$$

$$\text{and initially } \bar{\rho} = (1, 0, 0,\ldots, 0)$$

This form of the habit former will operate cyclically. At each iteration, the value of the i'th component is set equal to the value of the (i-1)'th component. Thus at each interaction the next habit will be taken. Gorn calls this form of the habit learner the "turn-taker" processor.[22]

It is evident why the term habit-former is applied to this processor. The modifications are made based upon the results of the result of the previous selection and not on the basis of and external experience. If the T operators were applied on the basis of experience gained, the processor would exhibit learning in the Gorn sense.

In order to see why the learning processor is more desirable than the habit-former, one may point out that the old saying "experience is the best teacher" is unreliable, because

experience without evaluation and/or guidance results in habit-forming that is non-goal directed. The present system incorporates the learning feature in its implementation.

However, there still is the function of adaptation that must be considered. With an adaptation capability, the system is able to scan what it has learned and from it make a guidance judgment. Thus the adaptive system can be pictured monitoring the learning system and directing it. This is represented in Fig. 4-3.

4.7 A Working Definition of an Adaptive Process

In the usual adaptive control system, there is some physical quantity that can be used in a mathematical optimization process. For example, in a manufacturing process, the cost per unit item produced can be computed and used to modify the generating function. Along this line, the following working definition is introduced.

> Adaptive process: A process that can monitor the operation of a system and on the basis of its performance, provide modifications that improve operations.

For each input, an output is produced and evaluated by the user. The system reacts to this evaluation and "scores" itself, determining how to modify the internal selection procedure. The adaptive monitor sits above it all and draws inferences concerning system performance.

4.8 The Final Form of the Adaptive Process

The development of the system was undertaken with two

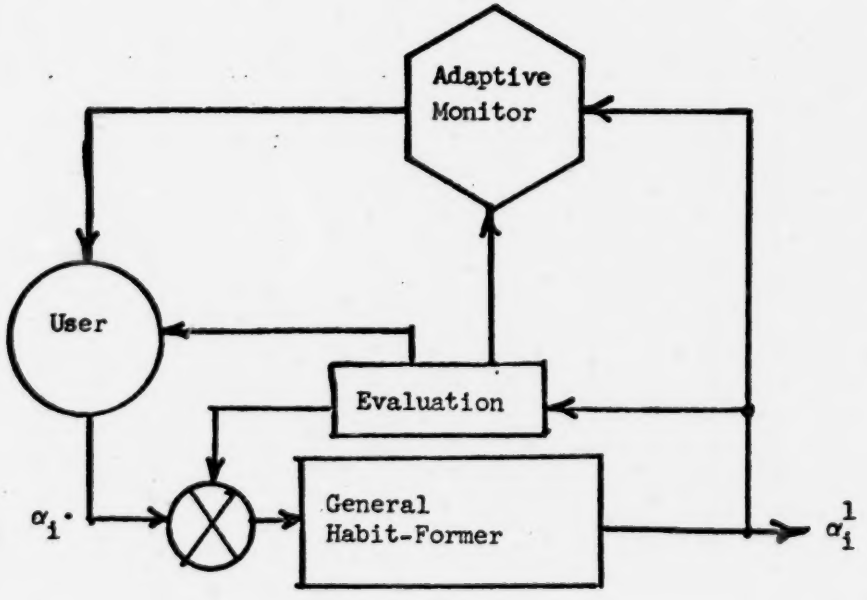


Figure 4-3 - General Habit-former with Learning and an Adaptive Monitor

requirements imposed:

(1) The system must learn to react to the presentation of an index set in a manner that is dependent upon the history of the interaction and in the light of the data set at the disposal of the machine.

(2) The system must be able adaptively to infer, from the interaction, general patterns and propose suitable measures to improve the pattern.

The manner in which the system fulfills these two requirements is covered in the two sections following.

4.9 Interaction-by-Interaction Learning

The system has at its disposal a set of microthesauri upon which its decisions are made. Initially, the system considers all microthesauri to be equal and assigns the same positive initial gain to all microthesauri. The system employs the current value of the score at each stage of the interaction as a "gain" for computing an inclusion number for each new suggested term found by the system. To arrive at the inclusion score, the system computes a set of contributions $C_a(t_j,t_k)$. Each contribution is formed by multiplying the gain of microthesaurus a, by the weight of the ordered pair (t_j,t_k) in it, where t_j is in the user's index set and t_k is a term with a non-zero weighted relation between it and t_k in microthesaurus a.

$$C_a(t_j,t_k) = G_a(N) \cdot w_a(t_j,t_k) \qquad 4\text{-}6$$

Where: $G_a(N)$ is the gain of microthesauri M_a after N interactions.

$w_a(t_j,t_k)$ is the weight of the ordered pair (t_j,t_k) in M_a.

t_j is in the user's index set and t_k is not.

The inclusion score $I_a(t_k)$ is obtained by summing over all of the j in the user's index set. Thus

$$I_a(t_k) = \sum_j C_a(t_j,t_k) \qquad 4\text{-}7$$

In order to be accepted as a proposed index term, at least one of the "a" values of $I_a(t_k)$ must be greater than a threshold T. Thus, if t_k is to be in M_N (the N'th machine response) then:

$$t_k \in M_N \Longleftrightarrow (\exists, a)(I_a(t_k) > t) \qquad 4\text{-}8$$

When the user reacts to the M_N, new gains are computed. The gains are increased by some reward R:

$$G_a(N+1) = G_a(N) + R \Longleftrightarrow I_a(t_k) > T \;\&\; t_k \in U(N+1) \qquad 4\text{-}9$$

For those terms proposed and rejected by the user, the system reduces the gains by some penalty P:

$$G_a(N+1) = G_a(N) - P \Longleftrightarrow I_a(t_k) > T \;\&\; t_k \notin U(N+1) \qquad 4\text{-}10$$

Normally the values of T, R, P, and the initial gain, G, will be parameters of the system and are easily changed by the

system operator. (See Section 4.12 for implementation.)

4.10 The Interest Profile

The values of the $G_a(N)$ represent a score of the various microthesauri after N interactions, and they are used by the system to remember which areas of specialization to stress. The set of gains are considered to be a representation of how the user is progressing in his search; for it is through the distribution of the gains that he is led to the speciality areas that interest him. Thus, the set of gains constitute an interest profile of the current user with respect to the current state of this interaction. This interest profile can be observed by the adaptive monitor which can then given direction to the user.

In Section 1.3, the nature of the adaptive process was introduced. In the next section, the adaptive algorithm is developed.

4.11 An Algorithm for Rendering Adaptive Assistance

As the interaction between the user and the system proceeds, the system constructs an interest profile of the user. Conceptually, all of the profiles can be observed as entries in a table. This is illustrated in Table 4-1. The a'th component of the interest profile after the j'th interaction is defined as $G_a(j)$. The value of a runs from 1 to A where there are "A" microthesauri. The j'th column of the table represents the state of the profile after j interactions, while the a'th row represents the history of the interest felt in the a'th speciality area.

In order to consider the three criteria of Section 1.3,

profile components	INTERACTIONS 1	2	...	j	...	N
G_1	$G_1(1)$	$G_1(2)$	...	$G_1(j)$	...	$G_1(N)$
G_2	$G_2(1)$	$G_2(2)$	...	$G_2(j)$	...	$G_2(N)$
G_3	$G_3(1)$	$G_3(2)$	...	$G_3(j)$	...	$G_3(N)$
.						
.						
.						
G_a	$G_a(1)$	$G_a(2)$	...	$G_a(j)$	...	$G_a(N)$
.						
.						
G_A	$G_A(1)$	$G_A(2)$	...	$G_A(j)$	...	$G_A(N)$

TABLE 4.1 - Table of Interest Profile

assume that the interaction is N and the components of the profile are arranged in decreasing order of magnitude.

Let this sequence of descending magnitude be G:

$$G = G_{i_1}(N),\ G_{i_2}(N),\ G_{i_3}(N),\ldots,G_{i_A}(N) \qquad 4\text{-}11$$

Where: there are "A" microthesauri.

$$G_{i_k}(N) - G_{i_{k+1}}(N) = df\ \Delta_k \geq 0 \qquad k = 1 \text{ to } (a-1)$$

The tendency for the interest profile to centralize is established when in the sequence G, first a small number δ of profile components exists such that the differences between them is small, and second there is a large difference between the last profile component in the set and the first component that is next in G. Symbolically:

$$1)\ \Delta_k \leq D_1 \text{ for all } k \leq \delta \qquad \text{where: } D_1 << D_2$$

$$2)\ \Delta_\delta \geq D_2 \qquad \delta \leq D_3 \qquad 4\text{-}12$$

Naturally the tendency for no centralization in the interest profile is recognized when after N iterations, the above conditions have not been met; there then will be no large spread in the differences. Symbolically:

$$1)\ D_1 < \Delta_k < D_2 \qquad 4\text{-}13$$

An oscillation in the set of microthesauri included in the high interest group will show the changing interest of the

user. Since the deletion of members from the high interest group is taken as a positive sign concerning the course of the interaction, the inclusion of new members is to be taken as a negative sign. Since it is possible for one microthesaurus to be dropped from the high interest group and another added on the same interaction, the test for this condition requires that the identity of the members of the high interest group be known. The implementation of this would involve denoting the microthesauri in the high interest group in the interest profile. Then after the conditions of Eq. 4-12 are satisfied, and a high interest group has been formed, any new addition of a microthesaurus to the group should be pointed out to the user.

4.12 An Extension to the Adaptive Algorithm

In many applications of information retrieval not only must one ascertain the extent of interest a user has in any given field of interest but it is also helpful to determine to what depth this interest extends. To some degree, the amount of interest can be determined through an examination of the terms employed by the user. In order to measure this depth, it is absolutely necessary to have prepared a structured classification of the terms that gives a generic-specific relationship between the terms in the system vocabulary. If this information were made available, then the technique of evaluating the depth of interest would be simple. Every time a term is rejected for a more detailed term or terms, the interest profile component in that area would also have its depth value augmented. The profile components then

would be constructed of an ordered pair of numbers, the first giving the value of the profile gain and the second giving the level of specifity. It must be emphasized that the useful development of such a technique would rest squarely on the data base, and there is not agreement among workers in the field as to the validity of generating a generic-specific table. In defense of the above technqiue, it may be pointed out that the user need not be aware of the system's viewpoint of the generic-specific relationship of any term pairs. The main force of the technique is that it is to be used on an averaging basis and does not need to rest on the validity of any one term pair. Continuing research at MSISL has shown the validity of generating the generic-specific classifications automatically. These are discussed below.

4.13 The Implementation of the Interactive Process

The interactive process has been implemented on the University of Pennsylvania 7040-PDP8 computer complex and the main programs are written in L6 programming language. The use of L6 and the set of programs that are used to perform the housekeeping functions of the system are discussed in Appendix A. Figure 4-4 is a copy of an actual interaction. The VECTOR editor program for the PDP8 computer was employed. The interaction proceeds when the L6 program requests the user to ENTER DATA. In this case, the user replied by typing TAPEIN < >. The symbols "<" and ">" are the VECTOR editor's END OF MESSAGE signal, "EOM", and the message is sent to the L6 program in the 7040 computer. TAPEIN causes the set of microthesauri to be read into the 7040

```
STANDBY.

ENTER DATA

     TAPEIN<>

TAPEIN

ENTER DATA

     PARAMETER
     REWARD
     4<>

PARAMETER

ENTER DATA

     SET
     OPERATING
     SYSTEM<>

SET

THE FOLLOWING WORDS ARE YOUR INDEX SET

OPERATING

SYSTEM

THE FOLLOWING TERMS CAN BE ADDED TO YOUR INDEX SET

ATLAS

MODULES

PRINTERS

TRANSMISSION

TURNAROUND

ONLINE

COMMAND

MONITOR

TIME-SHARING

DO YOU WISH TO ADD MORE TERMS
```

Fig. 4-4a - A Sample Dialogue

```
ENTER DATA

     YES<>

ENTER DATA

     TIME-SHARING
     COMMAND
     MONITOR<>

DO YOU WISH TO ADD MORE TERMS

ENTER DATA

     NO<>

ENTER DATA

     SET<>

SET

THE FOLLOWING WORDS ARE YOUR INDEX SET

OPERATING

SYSTEM

COMMAND

MONITOR

TIME-SHARING

THE FOLLOWING TERMS CAN BE ADDED TO YOUR INDEX SET

TURNAROUND

ONLINE

DEBUGGING

INTERRUPT

SUPERVISOR

REAL-TIME

BATCH

EXECUTIVE
```

Fig. 4-4b - A Sample Dialogue
(Continued)

```
DO YOU WISH TO ADD MORE TERMS

ENTER DATA

     YES<>

ENTER DATA

     EXECUTIVE
     SUPERVISOR<>

DO YOU WISH TO ADD MORE TERMS

ENTER DATA

     NO<>

ENTER DATA

     PARAMETER
     PROFILE<>

PARAMETER

PROFIL

000001

000020

000002
```

Fig. 4-4c - A Sample Dialogue

(Continued)

```
ENTER DATA
     TEST<>
TEST
THE FOLLOWING WORDS ARE YOUR INDEX SET
OPERATING SYSTEM
COMMAND
MONITOR
TIME-SHARING
SUPERVISOR
EXECUTIVE
ENTER DATA
     PARAMETER
     PROFILE<>
PARAMETER
PROFIL
000005
000005
000005
```

Fig. 4-4d - A Sample Dialogue

(Continued)

```
ENTER DATA

     SET
     OPERATING
     SYSTEM<>

SET

THE FOLLOWING WORDS ARE YOUR INDEX SET

OPERATING

SYSTEM

THE FOLLOWING TERMS CAN BE ADDED TO YOUR INDEX SET

ATLAS

MODULES

PRINTERS

TRANSMISSION

TURNAROUND

ONLINE

COMMAND

MONITOR

TIME-SHARING

DO YOU WISH TO ADD MORE TERMS

ENTER DATA

     YES<>
```

Fig. 4-4e - A Sample Dialogue

(Continued)

```
ENTER DATA

     TRANSMISSION
     MODULES<>

DO YOU WISH TO ADD MORE TERMS

ENTER DATA

     NO<>

ENTER DATA

     SET<>

SET

THE FOLLOWING WORDS ARE YOUR INDEX SET

OPERATING

SYSTEM

MODULES

TRANSMISSION

THE FOLLOWING TERMS CAN BE ADDED TO YOUR INDEX SET

ATLAS

PRINTERS

TURNAROUND

ONLINE

PARAMETERS

I/O

DEVICES

REGISTER

SIMULATION

DO YOU WISH TO ADD MORE TERMS
```

Fig. 4-4f - A Sample Dialogue
(Continued)

```
ENTER DATA

     YES<>

ENTER DATA

     I/O
     DEVICES<>

DO YOU WISH TO ADD MORE TERMS

ENTER DATA

     NO<>

ENTER DATA

     PARAMETER
     PROFILE<>

PARAMETER

PROFIL

000016

000001

000001

ENTER DATA

     TEST<>

TEST

THE FOLLOWING WORDS ARE YOUR INDEX SET

OPERATING

SYSTEM

MODULES

TRANSMISSION

I/O

DEVICES
```

Fig. 4-4 g - A Sample Dialogue
(Continued)

and the structure is constructed*. The L6 echoes the function to show what is being performed in the system.

After completing the construction of the microthesauri, the system again types ENTER DATA. This time the user types:

PARAMETER

REWARD

4 < >

This function sets the level of REWARD parameter to 4, and this is how much the interest profile will be augmented for each accepted term. The user then initiates the interactive dialogue by typing SET followed by the proposed index terms "operating" and "system". The system given OPERATE and SYSTEM as the index terms will, after consulting the microthesauri, propose the set of terms that can be added to the index set. Since this is the first stage of interaction, the values of the interest profile are all the same (they are all set to 000005). The first four terms suggested as possible additions were derived from the hardware microthesausus. TURNAROUND came from both the software and applications microthesaurus, and the last four terms came from the software microthesaurus. There is no implied ordering in the

---

* For the data of the present stage of implementation consisting of about 800 terms and 1770 entries in three microthesauri, the time elapsed to read the 1770 entries from tape and construct the three microthesauri was about 100 seconds. The might give insight into the running speed of the L6 system. In general, for the shorter functions there is no appreciable elapsed time between the entry of the function and the printing of the results. It would seem that the major processing limitation is the data link between the 7040 and the PDP8 to the teletypewriter. The employment of a CRT display and a high speed data link would speed up the process.

terms within a speciality. The system then asks if any of the suggested terms should be added to the index set. The user then types YES.

In the example, the user included three of the suggested terms in the index set. They are TIME-SHARING, COMMAND, and MONITOR. They may indicate that the user is interested in command monitors or operating systems for a time-sharing environment. The user wishes to continue the interaction and signifies this by typing SET < > to indicate that he does not choose to add any more of his own terms at that time. Again the system prints the current index set which now includes the new terms as well as the original ones. The suggested terms that are now proposed by the system are of a different character and seem to be related more nearly to the subject of the present example. None of the hardware terms is suggested at this time, showing that the system has learned that they are not of interest to the user. The user then chooses two synonyms of Operating Systems EXECUTIVE and SUPERVISOR. These synonyms were not strongly enough related to the original set to be included in the first stage of interaction; but after the profile was modified and the additional terms included, they were discovered by the system.

The user then requested a PROFILE of his request. The result shows that the gain for the area of hardware is reduced to one, software's has grown to 20, and application's is reduced to two. The final function of TEST < > produces a listing of the final index set and restores the system to the initial conditions.

The user then requested a profile to verify that it had been reset. The next interaction repeated the same initial index set only this time the hardware terms "TRANSMISSION" and "MODULES" were selected. The dialogue proceeded to the conclusion. The final profile is now given as 000016, 000001, and 000001 in hardware, software, and applications.

If at any time, the user proposes a word that is not a term in the system vocabulary, then the system will turn the word back with an appropriate comment. All terms that are proposed by the system and rejected by the user will reduce the effect of the microthesaurus that generated the term. The user is free to add terms of his choice to the index set at each stage of the interaction. He may also eliminate any terms from the index set by typing:

DELETE

term < >

If a term that is not in the index set is the argument of a DELETE function, then DELETE will fail and the system will type an appropriate comment. Deletions of terms from the index set do not affect the state of the interest profile.

At any point that the user is in control of the system, he may employ any of the system functions. The interaction proceeds from one execution of TEST to the next.

The technique used to identify interesting words was selected after numerous discussions at the Moore School Information Systems Laboratory of the properties that interesting and uninteresting terms should possess. Because this method had never been fully verified, subjective changes were made in the list of words chosen or rejected. With the ultimate aim of clumping in mind, it was decided to eliminate any word that occurred in more than 6-7% of the documents. Additional changes were made solely on the basis of the author's subjective judgement.

In the above discussions, the word 'word' was used rather loosely. Because many different words referred to the same concept (e.g., computer, computers), these concepts were identified and the statistics were computed for each 'merged group' of words. At the end of the selection process, 809 words had been chosen and they comprised 406 groups.

4.14 Clumping - The Techniques Considered

Numerous methods for clumping have been described in the literature, some as early as 1958. With the approaches varying greatly from one to the next, a careful comparison of techniques was required.

Five major considerations determined the applicability of any method. They were:

1) Type of data permitted,
2) Programming difficulty,
3) Amount of memory needed,
4) Operating speed, and
5) Relevance of results for intended usage.

In their article, Dale and Dale[1] describe a method which anticipates a data base of key words--words representative of a field and chosen independently of the particular documents in the data base. Associated with each word is a vector with elements of zero or one. A nonzero entry indicates the occurrence of that word in the corresponding document. The number of ones in the intersection of their vectors determines the association of two words and is called the "connection" of the two words. The "bias of a word to any subset" of key words is defined as the total connection of the word to all members of the subset, minus the total connection of the word to all nonmembers of the subset (it may be positive or negative). A subset is a clump if all members of that subset have a positive or zero bias to the subset and if all nonmembers have a negative bias to the subset. In short, a clump is a set of words whose members are more highly related to themselves as a group than they are related to the nonmembers of a group, and whose nonmembers are more highly related to themselves than they are to the members.

Three basic objections to this technique seem apparent. The subject vocabulary which concerns us comes from the text of a collection of abstracts and is thus quite different from the key word vocabulary which the above technique anticipates. Perhaps a major redefinition of the connection measure would solve this problem, but this could not be decided without much additional study. The machine time needed to run the clumping program for all interesting initial partitions seems extremely high. Possibly

the most crucial criticism is that even when the clumps are refined, they still overlap greatly and are therefore of little use to us.

Investigators at the Cambridge Language Research Unit have adopted a different approach[4]. Object-property information is processed to form a 'similarity matrix'. Subsequently, this matrix is manipulated to identify groups of objects sharing common properties.

Input to this technique consists of properties, each followed by a list of objects possessing that property; no mention is made of words, documents, etc. The similarity of two objects is defined as the ratio of the number of properties they share to the total number of distinct properties they possess. A cohesion function determines when clumps have been formed, a clump being a local minimum of the cohesion function. If SAB represents the sum of the similarities between members of A and those of B, and C is the potential clump, and $\bar{C}$ its complement, then one of the suggested cohesion functions is given by $\frac{(S\bar{C}C)^2}{SCC \times S\bar{C}\bar{C}}$. This measure emphasizes "coherence", the extent to which clump members are interrelated. To emphasize "separateness", the extent to which clump members are distinct from the remaining objects, $\frac{S\bar{C}C}{SCC} \times \frac{(NC)^2 - NC}{SCC}$ is the recommended cohesion function, where NC is the number of objects in C. The clump finding algorithm consists of an interative scan of the object list to see whether shifting an object in or out of the potential clump will reduce the current value of the cohesion function. This process must terminate in a stable partition of the object universe. If the total number of objects is not too

large, each object may be taken as a prospective clump and we may attempt to grow a clump around it. Another possibility is to select pairs or triples of highly connected elements as starting groups.

Rapid identification of clumps (one every 1.5 seconds) makes this method highly desirable. The freedom to choose a cohesion function which allows the coherence and separateness components to be manipulated independently is also a significant factor. Certain clumps, however, seem likely to be missed entirely as the search strategy is order-dependent. Careful consideration was given to adapting our data to the input form required but the concept of properties possessed by data elements was still inappropriate to the problem at hand. This difficulty alone was sufficient to exclude the above approach to the clumping task. The distinction between separateness and coherence remains valid for other techniques and should be considered when analyzing them.

R. M. Needham, also a member of the Cambridge Language Research Unit, describes another technique[6]. The particular problem of how to identify a term as a suitable substitute for another term in a retrieval request forms the basis of his approach.

A measure of association between two words, the connection, is defined as the ratio

$$\frac{\text{Number of documents in which the terms co-occur}}{\text{Number of documents in which at least one occurs}} .$$

The N by N connection matrix A, where N is the number of words,

has as its entries the appropriate connection values. By convention, diagonal elements are set equal to zero. An N-vector $\bar{V}$, with elements of ± 1, specifies a group of terms, +1 denoting a member of the group, -1 a nonmember. A clump is defined as any subset of terms for which the sum of the connection of any member to all other members exceeds the sum of its connection to all nonmembers, and for which any nonmember's connection sum to all other nonmembers exceeds its connection sum to all members. (See Figure 2.)

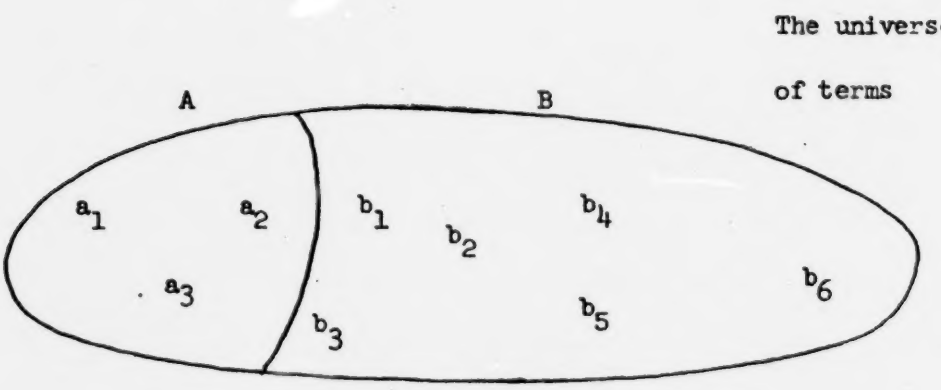


Figure 2 - a clump

A is a clump if each a is more closely connected to all other a's together than it is to all b's together, and if each b is more closely connected to all other b's than it is to all a's.

With $\bar{1}$ denoting a vector all of whose elements are +1, cohesion is defined as the ratio

$$\frac{(\bar{1}'A\bar{1} - \bar{V}'A\bar{V})}{(\bar{1}'A\bar{1} + \bar{V}'A\bar{V})}$$

where the prime indicates the transpose. A clump is a local minimum of cohesion, that is moving any element into or out of the clump increases the cohesion. Starting from some initial partition of the universe, specified by $\bar{U}$, each element is tested sequentially and is transferred whenever such a movement would decrease the cohesion. The cohesion does not have to be computed at each stage. Let $A\bar{U} = R\bar{U}$ where R is some diagonal matrix. If any element of R is negative, we make it positive by reversing the sign of the corresponding row of $\bar{U}$. This forces the other elements of R to be changed, but each such step reduces the cohesion until a clump is found.

Eventually, the sample size will cause available core storage to be exceeded, even in the largest machines. Although this problem is avoided by storing the connection matrix on magnetic tape, the running time remains very reasonable.

As is the case with every technique discussed so far, there is no guarantee that all clumps will be found.

The article gives no indication of the extent to which clumps overlap; without such information it is somewhat difficult to evaluate this method.

The only major objection to this technique, and it is relative to the others considered, is that the actual algorithm of this technique fails to make sense intuitively. In some of the other

methods it is easier to see the role each step plays in identifying closely knit groups. If a better technique were not available, this objection could be ignored.

The last technique, developed by Stiles and Salisbury[7], consists of "growing a clump" by adding one element at a time. Initially, the vocabulary is divided into a two member group and its complement. A rather simple and direct method may be used to identify the starting group. At each stage in the clumping process, that member of the complement yielding the highest value of the measure which the authors specify is added to the existing group to form a new one. This process is repeated until no new word seems to fit in well with the current clump. A new starting partition is determined and the search for the next clump begins.

A matrix of the correlation among all words in the subject vocabulary is needed in selecting the initial groups and in deciding when a word should be added to the present clump. The association between words i and j is given by:

$$A_{ij} = \frac{(C_{ij})D - (C_{ii})(C_{jj}) - D/2}{\sqrt{(C_{ii})(C_{jj})(D-C_{ii})(D-C_{jj})}}$$

where C_{ij} is the number of documents in which i and j both occur (i.e., co-occur) and D is the total number of documents. C_{ii} is just the number of documents in which word i occurs.

The measure used to determine which word to add to the clump is called the B-coefficient and is defined as "100 times the ratio

of the average of the intercorrelations among the variables within a group to their average correlation with all remaining variables" *.

Three hypotheses must be satisfied for this technique to be applicable:

1) The vocabulary must have a common factor;
2) There must exist at least one group of words S which possess some common property not shared by the remaining words;
3) Given a method of measuring the association between any two words of the vocabulary, the association between any two members of S is higher than the association between any member of S and any nonmember.

It was immediately apparent that our data base satisfied the first two hypotheses. Satisfaction of the third requirement could not be verified, but intuitively it would have to be at least partially satisfied if the concept of clumps were to be at all meaningful. (Results did indeed verify this initial faith.)

The results obtained by the authors appeared to be appropriate for our intended use. Programming this technique promised to be very straightforward so long as the entire association matrix could be stored in core. Using auxiliary memory would drastically increase operating speed.

---

* Ref. [7], p. 5.

Intuitively, this technique made as much sense or perhaps more than any other. Thus, devising a technique to fit the bulk of the association matrix into core appeared to be the main problem. Storing only non-zero elements, using auxiliary tables to keep track of the coordinates and taking the lowest values to be zero, permitted most of the matrix to be stored in core.

This technique satisfied the five requirements listed previously. Together with its intuitive clearness this fact accounted for its being selected.

4.15 Implementation

A clumping technique using the B-coefficient was implemented on an IBM 7040 computer. Time considerations required that the entire 406 x 406 association matrix, AM, be stored in core; however, only 23,000 memory locations were available. Storing just the 25,000 non-zero entries offered some help but each entry appeared to need two coordinates (row, column) to identify it. Noticing that for entries from a given row the first coordinate was always the same, we found a way to reduce the core required (by almost 33%) by eliminating this repetition. In addition to a main array, A, two auxiliary arrays were used to store 1) the number of non-zero entries in each row of AM and 2) the starting position within A of all information relating to a given row of AM.

This approach proved sufficiently useful to warrant further explanation. The 2-dimensional association matrix, AM, is

stored sequentially in the 1-dimensional array A. Corresponding to each non-zero element in AM, a double entry is made in A consisting of the coordinate of the column in AM followed by the actual AM entry. A second array, START, contains as its ith element the starting location within A of all information relating to row i of AM. The array NUM contains in a similar fashion the number of non-zero entries in the corresponding row of AM.

Consider the following example:

AM				
	0	0	6	0
	0	0	9	0
	6	9	0	7
	0	0	7	0

A:	3	6	3	9	1	6	2	9	4	7	3	7
START:	1	3	5	11								
NUM:	1	1	3	1								

To find AM(3,4) from A, START, and NUM, we first get START(3) and NUM(3). Searching through NUM(3) pairs in A beginning with A(START(3)), we either find the number 4 in which case the next location in A contains AM(3,4) or if we do not find it, the value must be zero.

The following chart shows the difference between the three methods. AM is assumed to be N x N; K% of the entries are non-zero.

	Storing All AM	Storing Non-Zero Elements	Storing Non-Zero Elements; Auxiliary Arrays
General Formula	N^2	$(.03K) \times N^2$	$(.02K) \times N^2 + 2N$
Our Data Base N=406, K=15	165,000	74,000	50,000
Typical Future Case N=2000, K=10	4,000,000	1,200,000	804,000

Table 1

Number of Core Locations Needed

Since our original matrix is symmetric, only entries above or below the diagonal need be stored regardless of which method is selected. However, implementing this promised to increase the programming problems and the running time. We decided to store as many non-zero elements (from both below and above the diagonal) as space permitted. This required setting all entries below a certain value equal to zero. In the future it seems desirable to write the clumping program to take advantage of the symmetry and store only elements from below the diagonal. Results to date indicate that the extra operating time could be afforded.

The B-coefficient was defined as "100 times the ratio of the average of the intercorrelations among the variables within a group to their average correlation with all remaining variables".

With $AM(V_i,V_j)$ denoting the correlation between variables V_i and V_j, G representing the group, and $\bar{G}$ its complement (for a total of N elements), we can express the B-coefficient as follows:

$$B(k) = 100 \frac{\sum_{j=1}^{k} \sum_{i<j} AM(V_i,V_j)/(k(k-1)/2)}{\sum_{j=1}^{k} \sum_{\ell=k+1}^{N} AM(V_j,V_\ell)/k(N-k)}$$

where $V_j \in G \qquad j = 1,\ldots,k-1$

$V_\ell \in \bar{G} \qquad \ell = k+1,\ldots,N$

and V_k is the prospective addition from $\bar{G}$ to G.

Denoting the double sum in the numerator by P(k) where k indicates the element about to be added, and the double sum in the denominator by T(k) we can significantly reduce the number of calculations necessary at each stage as follows:

$$P(k) = \sum_{j=1}^{k} \sum_{i<j} AM(V_i,V_j) \qquad \text{(leaving out } AM(V_i,V_j) \text{ from the intermediate steps)}$$

$$= \sum_{j=1}^{k-1} \sum_{i<j} + \sum_{(j=k)} \sum_{i=1}^{k-1}$$

$$= P(k-1) + AKG(k)$$

where $AKG(k) = \sum_{i=1}^{k-1} AM(V_k,V_i)$ is the total association (A) of the prospective addition (K) to the group (G).

$$T(k) = \sum_{i=1}^{k} \sum_{j=k+1}^{N} AM(V_i, V_j)$$

$$= \left(\sum_{i=1}^{k-1} + \sum_{(i=k)} \right) \left(\sum_{j=1}^{N} - \sum_{j=1}^{k-1} - \sum_{(j=k)} \right)$$

$$= \sum_{i=1}^{k-1} \sum_{j=1}^{N} - \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} - \sum_{\substack{i=1 \\ (j=k)}}^{k-1} + \sum_{\substack{j=1 \\ (i=k)}}^{N} - \sum_{\substack{j=1 \\ (i=k)}}^{k-1} - \sum_{(j=k)}$$

$$= \sum_{i=1}^{k-1} \sum_{j=k}^{N} - 2 \sum_{\substack{i=1 \\ (j=k)}}^{k-1} + \sum_{\substack{j=1 \\ (i=k)}}^{N} \quad \text{(using } AM(V_k, V_k) = 0\text{)}$$

$$= T(k-1) - 2\,AKG(k) + AKT(k)$$

where $AKT(k) = \sum_{j=1}^{N} AM(V_k, V_j)$ is the total association (A) of the prospective addition (K) to the total (T) collection of elements.

B(k) can now be expressed as

$$\frac{200(N-k)\,P(k)}{(k-1)\,T(k)}$$

where $P(k) = P(k-1) + AKG(k)$

and $T(k) = T(k-1) - 2\,AKG(k) + AKT(k)$.

Instead of computing 2 double sums at each stage, we must perform a few immediate calculations plus some simple updating when an element is actually added to G (assuming we store P,T,AKG and AKT). A quick calculation will show that for each clump of 40 members over 100 million additions will be saved, a significant and necessary reduction even on the fastest computers.

The association measure as described earlier was rather clumsy to use. Noticing that the number of documents in which words occur is much smaller than the total number of documents, we may write

$$A_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}\ C_{jj}}}$$

where C_{ij}, C_{ii}, C_{jj} have the same meaning as before.

4.16 Results

It was our intention to experiment with different association measures and we in fact tried:

1) $A_{ij} = \dfrac{C_{ij}}{\sqrt{(C_{ii})(C_{jj})}}$ (approximation to Stiles' measure)

2) $A^{*}_{ij} = \dfrac{C_{ij}}{\text{maximum } (C_{ii}, C_{jj})}$

3) $A^{**}_{ij} = \dfrac{C_{ij}}{\text{minimum } (C_{ii}, C_{jj})}$

The latter two were chosen in an attempt to bracket all likely association measures. (All likely measures were expected to give

values between A** and A*.) Most other measures could be produced by combining these two in various ways. According to an article by Jones and Curtice[5], A* would emphasize particular terms while A** would emphasize general terms. We did not attempt to verify this because of the nature of our data base, i.e., its being spread out over many subareas. This is sufficiently interesting to warrant further investigation. It would be wrong to draw any conclusions on this subject from the clumps included here.

Using the B-coefficient to clump, we obtained the results shown below (Figure 3). The first four words are highly related to one another but there are some which have been suggested as members of the clump yet do not appear to be related to any other member of the clump. Table 2 may provide some insight into what has happened. Apparently, words with a small total association have been added to the clump even though they have a low or zero association with it.

It would be helpful at this point to consider how a word may be added to an existing clump:

1) The word has a high association with the clump (and a small association with the complement).
2) The word has a small total association thereby giving a large value to its B-coefficient, independent of its association with the clump.

A small total association indicates that we do not have enough information about this word to properly place it in one clump or

Benzophenone		100	33	40			23		
Ketyl	100		33	40			23		
Transient	33	33			25		7		
Triacetate	40	40				20	15	20	20
Disk			25						
Solvation				20				33	
Species	23	23	7	15					
Adiabatic				20		33			
Log									
Acetyl				20					
Soil									
Thermocouple									
Landing									
Ablator									
Prefabricated									

Figure 3

Association Matrix #1 using A^*_{ij}

Without Total Association Criterion

Note: The words are listed in the order in which they were added to the clump.

Benzophenone	25000	Log	400
Ketyl	25000	Acetyl	7400
Transient	15300	Soil	600
Triacetate	20600	Thermocouple	1000
Disk	7900	Landing	2100
Solvation	6800	Ablator	2400
Species	23900	Prefabricated	2500
Adiabatic	18600		

Table 2

The Total Association of a word with the entire vocabulary, i.e., the sum of the association of a word with all other words.

another. One might be tempted to use a word's frequency to determine whether to exclude the word; however, a group of low frequency words could still be meaningfully clumped if they always occurred in the same abstracts. Therefore, we decided to exclude all words whose total association was less than 1/3 the maximum total association of any word thereby reducing our vocabulary by 33%. The exact fraction used was chosen somewhat arbitrarily. A small change would only effect borderline words; thus the choice does not appear critical. Except for the association matrix which led us to intr luce the total association criterion, all of the others have already taken this criterion into account. The effect can be seen by comparing Figure 3 with Figure 4 .

A word with a high association to an existing clump may not be added to the clump if the word has a high association with the complement. For some purposes this is a desirable restriction, but not in our case. To correct this, we introduced a modified B-coefficient. Using the same notation as before we now have $T(k) = T(k-1) + (AKT(k) - 2\ AKG(k))/H$. In the results which follow when we talk about the modified B-coefficient, we will mean a value of $H = 3$. An area for further investigation would be to determine whether there are a number of most desirable values for H, each one corresponding to some distinct type of clumping.

Figures 4 and 5 show the effect of the modification in the B-coefficient. In Figure 6 we see that changing the association measure to A_{ij} did not have any significant effect, while using measure A^{**}_{ij} produces the same initial clump but leads into a

Benzophenone		100	33	40	23			11							
Ketyl	100		33	40	23			11							
Transient	33	33			7			11							
Triacetate	40	40			15	20									
Species	23	23	7	15				7							6
Adiabatic				20			14								
Isothermal						14			14	12	12				
Decay	11	11	11		7				11			11		6	
Caprolactam							14	11		12	12	14		6	
Amide							12		12		12			6	
Sorption							12		12	12		12			
Potassium								11	14		12		42	26	20
Silane												42		20	26
Kcal								6	6	6		26	20		20
Siloxane					6							20	26	20	

Figure 4 - Association Matrix #1 using A^*_{ij} With Total Association Criterion

Note: Not every word belongs to the clump; in fact, the clump would probably stop after 'SPECIES'. A cutoff procedure will be discussed later.

Benzophenone		100	40	33	23	11		6		6	4	4		5	5
Ketyl	100		40	33	23	11		6		6	4	4		5	5
Triacetate	40	40			15		20	6						13	5
Transient	33	33			7	11									
Species	23	23	15	7		7		6	5	10	5	4		5	8
Decay	11	11		11	7			6	7	8	5				
Adiabatic			20												
Irradiated	6	6	6		6	6			20	12	16	9	5	7	5
Electron					5	7		20		20	20			7	
Radical	6	6			10	8		12	20		19	6	6		
Spectra	4	4			5	5		16	20	19		20	27	5	5
Absorbance	4	4			4			9		6	20		14	9	9
Infrared								5		6	27	14			
Cellulose	5	5	13		5			7	7		5	9			13
Color	5	5	5		8			5			5	9		13	

Figure 5 - Association Matrix #1 using A^{*}_{ij} Modified B-coefficient

Benzophenone		100	51	48	33	19	18	20	15	21	24	19	22		10
Ketyl	100		51	48	33	19	18	20	15	21	24	19	22		10
Triacetate	51	51		24		15	36	15	12	10	12		11	8	
Species	48	48	24		16	14	8	9	11	13	19	9	10		
Transient	33	33		16							8	19			10
Color	19	19	15	14			13			8			12		
Cellulose	18	18	36	8		13		8	8				12	17	
Irradiated	20	20	15	9			8		26	23	15	11	13	9	
Electron	15	15	12	11			8	26		23	21	18		10	
Spectra	21	21	10	13		8		23	23		22	16	21	13	12
Radical	24	24	12	19	8			15	21	22		18			15
Decay	19	19		9	19			11	18	16	18				17
Absorbance	22	22	11	10		12	12	13		21					
Graft			8				17	9	10	13					19
Polymethyl	10	10			10					12	15	17		19	

Figure 6 - Association Matrix #1 using A_{ij} Modified B-Coefficient

Absorbance		100	100	23	17	40	15	18			22	100	66	22	40	20
Benzophenone	100		100	100	66	66	66	66	66	33	100					
Ketyl	100	100		100.	66	66	66	66	66	33	100					
Species	23	100	100		23	40	15	15	23		30					
Color	17	66	66	23		40										37
Triacetate	40	66	66	40	40		100	40	40		40					
Cellulose	15	66	66	15		100								20		
Irradiated	18	66	66	15		40			34	22	34					
Electron		66	66	23		40		34		44	26				20	
Decay		33	33					22	44		44					
Spectra	22	100	100	30		40		34	26	44		100	66	20	40	
Chlorophyll	100										100		100	100	100	100
Nitromethane	66										66	100		66	66	66
Nitrogen	22						20				20	100	66		40	20
Pyridine	40								20		40	100	66	40		40
Pigment	20				37							100	66	20	40	

Figure 7 - Association Matrix #1 using A^{**}_{ij}

completely different one (Figure 7).

Before we can determine which measure gives the best results we must answer two questions:

1) How do we identify a clump?

2) How do we judge the quality of a clump?

As discussed earlier, a clump is a set of words which are highly related to one another. Some of these words, but not all, may have a high association with the complement. Stiles and Salisbury used the ratio of the current B-coefficient to the previous one to determine where a clump ends, but their results with this method and our own results indicate the need for another technique.

If the vocabulary were divisable into non-overlapping perfect clumps and if the words were ordered such that all words in clump one came first, etc. (as the clumping procedure would order them), the association matrix would be block diagonalized (Figure 8) with each block being a clump. In reality, some words belong in more than one clump and some members of a clump may not be related to every other member (i.e., some asterisks would represent zero entries). In Figure 9, we see two clumps (one consisting of W3, W5, W6, W7, and the other of W8, W9, W10) linked together by W1, W2, and W4. If there had been more overlap between the two clumps we would have identified them as a single clump.

The members of a clump should form a square block - with a small percentage of zero entries. Except for those words which also belong to other clumps, the words in this clump should have little association with other words. This technique is a visual

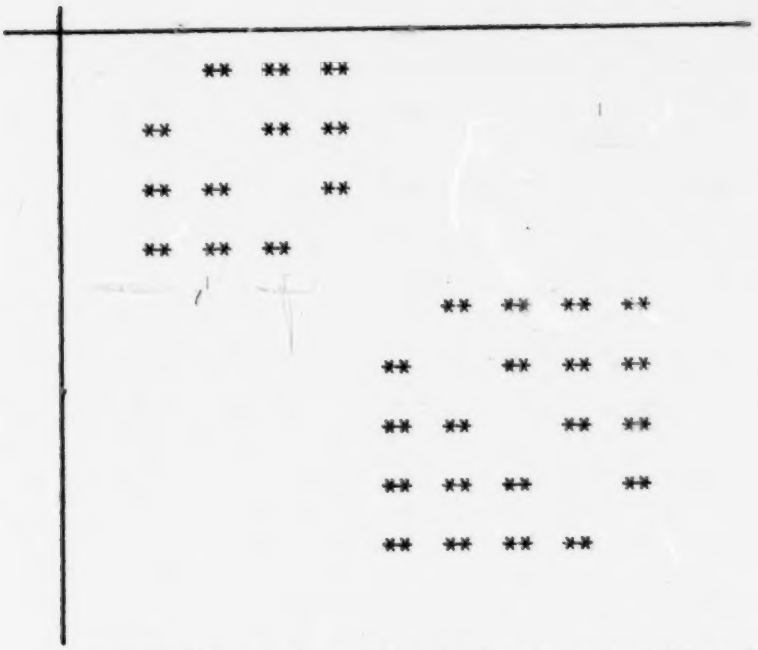


Figure 8 - A Block Diagonalized Association Matrix

W1		**	**			**	**		**	
W2	**		**	**	**		**		**	**
W3	**	**		**	**	**				
W4		**	**		**		**	**		**
W5		**	**	**		**	**			
W6	**		**		**		**			
W7	**	**		**	**	**				
W8				**					**	**
W9	**	**						**		**
W10		**		**				**	**	

Figure 9 - A Typical Association Matrix Where 2 Clumps are Linked by a Few Words

NOTE: ** represents non-zero values.

one and can be readily applied after having the clumping program produce a graph of the association matrix. In Figure 7, we would include all words through SPECTRA as members of the clump. ABSORBANCE and SPECTRA also belong to another clump which is forming. In Figure 10, we can see two subclumps with the words ENTHALPY, ENTROPY, and EQUILIBRIUM belonging to both. Changing the measure to A_{ij} (Figure 11) produces a different clump. More of the association matrix must be seen to determine where this clump ends. There are some instances where it is difficult to identify a clump (Figure 12). In general, this technique seemed to work well; most association matrices included within them some pattern which we were able to identify as a clump.

One way to judge any clumping technique would be to use the clumps it produces in Edwards' adaptive interface and allow a number of users to test them. This would not judge an individual clump but rather a set of clumps as a whole. This process would be insensitive to small improvements. Because of its elaborateness, we might best use this technique to verify some other approach.

A second method requires an expert in the field to analyze the various clumps to determine whether any good words have been left out or whether any words do not belong to the clump which they are in. However, the expert reflects the field in general and not the particular documents of our data base. His expertise may just destroy his usefulness in this situation.

Enthalpy		70	20	20	20	10	10	10	10	9	11	[illegible]	7	5	6
Entropy	70		20	20	20	10	10	10	10	9	[illegible]	[illegible]	7	5	6
Osmometer	20	20													
Divinylbenzene	20	20								9	23	17	15	10	6
Calorimeter	20	20				20	10	10	10	4					
Polyoxymeth-ylene	10	10			20		12	12	12	4					
Trioxan	10	10			10	12		14	12	9					
Adiabatic	10	10			10	12	14		12						
ATM	10	10			10	12	12	12		4					
Equilibrium	9	9		9	4	4	9		4		9	9	7	5	6
Exchange	11	11		23						9		23	19	23	10
Sodium	11	11		17						9	23		19	18	10
Cation	7	7		15						7	19	19		28	6
Ion	5	5		10						5	23	18	28		18
Aqueous	6	6		6						6	10	10	6	18	

Figure 10- Association Matrix #2 Using A^*_{ij} Modified B-coefficient

Enthalpy		70	25	15	15	10	12	11	13	10	8				
Entropy	70		25	15	15	10	12	11	13	10	8	13			
Divinylbenzene	25	25		39	29	26	32	14	17	19	15	8		16	9
Exchange	15	15	39		23	35	23	13	10	15	15			12	
Sodium	15	15	29	23		27	23	13	10		9			14	16
Ion	10	10	26	25	27		34	20		10	10			16	
Cation	12	12	32	23	23	34			8				8	15	8
Aqueous	11	11	14	13	13	20								10	8
Equilibrium	13	13	17	10	10		8			10	13	13	9		
Swell	10	10	19	15		10			10		14	20	24		
Cross-Linked	8	8	15	15	9	10			13	14		27	11	11	17
Network		13	8						13	20	27		9		
GEL							8		9	24	11	9		10	10
Acid			16	12	14	16	15	10			11		10		15
Hydroxide			9		16		8	8			17		10	15	

Figure 11- Association Matrix #2 using A_{ij} Modified B-coefficient

Container		40	16	7	8	6				5							
Bottle	40		10		4	4				4							
Package	16	10		11	7	9	5	20	7	10	8	17	5	6		5	
Cost	7		11		8	6	10	5		4	13	9	4	4	7	7	
Densities	8	4	7	8		5	4	4	6	8	15		12	9	7	8	
Extrude	6	4	9	6	5		26	11	19	7	7	8	6				
Inject			5	10	4	26		5	11	6	9		4				
Film			20	5	4	11	5		5	6		13	11			7	
Melt			7		6	19	11	5		4		6	4				
Polystyrene	5	4	10	4	8	7	6	6	4		16	4	9			7	
Foam			8	13	15	7	9			16		9	5	39	35	10	
Laminar			17	9		8		13	6	4	9		7	8		4	
Polypropylene			5	4	12	6	4	11	4	9	5	7		5			
Polyurethane			6	4	9						39	8	5		29	14	
Urethane				7	7						35			29		8	
Absorbance			5	7	8			7		7	10	4		14	8		

Figure 12- Association Matrix #3 using A^*_{ij} Modified B-Coefficient

Assuming the correctness of our theoretical description of what the association matrix of a clump should look like, we may also use the same method to determine the quality of a clump. Most members of a good clump should have a (high) non-zero association with one another. Each individual working with this clumping technique would decide for himself the amount of overlap between clumps that is permitted. Combining this visual effort with some expert analysis should provide the best solution.

In the discussions which follow, little mention is made of A^{**}_{ij}. Excluding the one clump using that measure which we have included, most of the others were rather poor, tending to form clumps of 3 or 4 words. For our data base, the A^{**}_{ij} measure frequently took on its maximum value. Perhaps its lack of spread explains the poor results.

In most cases where comparison was possible, A_{ij} gave better results than A^{*}_{ij} and the modified B-coefficient further improved the results. Comparisons of this sort were difficult at times because a difference between clumps of only one word affected all additional words. Each extra word in which clumps differed increased the likelihood of further differences. Figures 4 and 5 illustrate this point. In these same figures, we can also see the improvement made by modifying the B-coefficient. The first clump does not include many of the words which the second association matrix shows belong to the clump.

In Figures 13, 14, and 15 we have an example of the improvements obtained by first using the modified B-coefficient and then switching to A_{ij}.

Chlorophyll		66	40	20	15					7	8						
Nitromethane	66		40	20	15					7	8						
Pyridine	40	40		20	15			10		7	8						
Nitrogen	20	20	20		15	10				7	8	4		7		5	
Binder	15	15	15	15						7	12					5	
Diamine				10			20	10	11			20	17	7		5	
Polyimide						20						12					
Methylene			10			10						12	4				
Dampers						11											
Derivative	7	7	7	7	7						7						
Pigment	8	8	8	8	12					7				4		8	
Aromatic				4		20	12	12					20	12	4	4	
Polyamide						17		4				20		13	4	4	
Amine				7		7					4	12	13		14		
Isocyanate												4	4	14			
Phtnalate				5	5	5					8	4	4				

Figure 13- Association Matrix #4 using A^*_{ij}

Chlorophyll		66	40	20	15	7	8										
Nitromethane	66		40	20	15	7	8										
Pyridine	40	40		20	15	7	8										
Nitrogen	20	20	20		15	7	8		5		5						
Binder	15	15	15	15		7	12			4	5					7	4
Derivative	7	7	7	7	7		7	5	23				10				
Pigment	8	8	8	8	12	7		25	7	7	8	4		8			11
Color						5	25		13	9	5					5	9
Cellulose				5		23	7	13		31	10	7	7				
Acetate					4		7	9	31			9	4	7			
Phthalate				5	5		8	5	10			9	6	11	16		4
Plasticization							4		7	9	9		9	4			
Ester						10			7	4	6	9			6		6
Emulsion							8			7	11	4			5		4
Alkyd					7			5			16		6	5			22
Paint					4		11	9			4		6	4	22		

Figure 14 - Association Matrix #4 using A^*_{ij} Modified B-Coefficient

Chlorophyll		81	63	44	28	26	18								17	
Nitromethane	81		51	36	23	21	14			8					13	
Pyridine	63	51		28	18	16	11								10	
Nitrogen	44	36	28		12	11	8		10							
Pigment	28	23	18	12			13	31	9	9						
Derivative	26	21	16	11					27						9	
Absorbance	18	14	11	8	13			12	12		11	22	22	10	21	13
Color					31		12		13	10	15	19	19	14	8	
Cellulose				10	9	27	12	13		32	36	18	18	8		8
Acetate		8			9			10	32		13					
Triacetate							11	15	36	13		51	51	24	10	15
Benzophenone							22	19	18		51		100	48	21	20
Ketyl							22	19	18		51	100		48	21	20
Species							10	14	8		24	48	48		13	9
Sectra	17	13	10			9	21	8			10	21	21	13		23
Irradiated							13		8		15	20	20	9	23	

Figure 15 - Association Matrix #4 Using A_{ij} Modified B-coefficient

No mention has been made so far of the actual clumps this technique has produced and despite any argument to the contrary, the reader might feel himself deprived of important evidence. A few of the association matrices included in this report do not show a full clump or do not give enough information to determine where a clump ends. Others provide excellent examples. From Figure 7 we conclude that the words BENZOPHENONE, KETYL, SPECIES, COLOR, TRIACETATE, CELLULOSE, IRRADIATED, ELECTRON, DECAY plus ABSORBANCE and SPECTRA form a clump with the latter two terms also belonging to another clump which is forming (only partially shown).

In Figure 10 the 2 clumps which have formed are quite apparent, being tied together by ENTHALPY, ENTROPY, and EQUILIBRIUM, and perhaps also by OSMOMETER.

CHLOROPHYLL, NITROMETHANE, PYRIDINE, NITROGEN, PIGMENT, DERIVATIVE, ABSORBANCE, CELLULOSE, and SPECTRA all belong to one clump (Figure 15).

One final association matrix is shown in Figure 16. All words except the last 2 definitely belong to a single clump.

Isopropoxide		77	54	32	23	20	18	18	24	22	18	16	17	16		
Nitrile	77		42	24	17	16	14	14	18	17	14	12	13	12		
Acetone	54	42		17	12	11	10	13	13	18	15	13	9	8		
Anionic	32	24	17		11	9	12	29		16	13	19	12			27
Acrylonitrile	23	17	12	11		17	17		11			11		9		
Reactive	20	16	11	9	17		23	19	10	14	11	15	11	10		
Monomer	18	14	10	12	17	23		25	18	19		12	8	17	8	
Catalysis	18	14	13	29		19	25		25	23	17	12	9	12	14	10
Aluminum	24	18	13		11	10	18	25						9		
Cation	22	17	18	16		14	19	23			12	16	12			
Ethylene	18	14	15	13		11		17		12		13	17	11	37	
Radical	16	12	13	19	11	15	12	12		16	13					14
Oxide	17	13	9	12		11	8	9		12	17				18	
Mix	16	12	8		9	10	17	12	9		11				9	
Propylene							8	14			37		18	9		9
Stereoregularity				27				10				14			9	

Figure 16 - Association Matrix #5 Using A_{ij} Modified B-coefficient

5. ENGLISH AS A SEARCH LANGUAGE

5.1 Easy English

As mentioned in section 1, appreciation of man-machine communication problems led to the mechanization of a simplified natural language called Easy English for use for searchers of the Information System. The success of Easy English led to the development of the more complex "Real English" using complete grammar incorporated into the computer. These languages are described in this section.

Easy English is a plain command language designed to simplify dialogues between man and machine through a remote typewriter console. It is made up of readily recognized sentences of the English language, sentences which any layman might be expected to use in everyday requests for services or articles from a familiar source. Easy English has been developed as a command language for retrieval of documents from a computerized data base, specifically from the Moore School Information Systems Laboratory (MSISL) files. It is intended for all information retrieval systems using remote typewriter consoles in a conversational mode.

Easy English is imbedded in the MSISL retrieval program which provides computer-directed search, computer-aided editing, and other forms of computer assistance. The attached typewriter printout presents a typical man-machine conversation which illustrates Easy English along with a number of features of the Laboratory retrieval system. Note that the latter currently provides the option of translation of the Easy English request into Symbolic Command Language while searching the files; this is a convenience for those who might like to learn Symbolic Language on their own and use its shorter but more formal statements in place of Easy English.

Because Easy English is in fact real English, the only thing that the searcher needs to learn is that requests for information from the system should be formulated in the following syntactical form:

[Introductory Clause] [Document Clause] [Data Clause] .

The following sentences present five forms in which the same retrieval request can be phrased in Easy English:

(1) PLEASE LOCATE EVERYTHING WRITTEN BY ROBERT PERKINS ABOUT EASIAC OR PSEUDO-COMPUTERS BETWEEN 1955 AND 1959 < >

(2) COULD YOU FIND FOR ME SOMETHING CONTAINED IN THE REPOSITORY CONCERNING EASIAC OR PSEUDO-COMPUTERS THAT WAS AUTHORED BY ROBERT PERKINS AFTER 1954 AND BEFORE 1960 < >

(3) I NEED ALL THE AVAILABLE DOCUMENTS PUBLISHED DURING THE PERIOD 1955 TO 1959 BY ROBERT PERKINS ON THE SUBJECTS OF EASIAC OR PSEUDO-COMPUTERS < >

(4) WE'RE INTERESTED IN HAVING REFERENCES AND MATERIAL ON EITHER PSEUDO-COMPUTERS OR EASIAC AUTHORED BY ROBERT PERKINS FROM 1955 TO 1959 < >

(5) I WOULD LIKE YOU TO HELP ME OBTAIN INFORMATION FROM YOUR LIBRARY RELATED TO RASIAC OR PSEUDO-COMPUTERS AND WRITTEN BY ROBERT PERKINS IN THE YEARS 1955 THROUGH 1959 < >

Notice that despite the differences in vocabulary, all of these statements follow the same basic pattern; for example,

[COULD YOU FIND FOR ME] [SOMETHING CONTAINED IN THE REPOSITORY] [CONCERNING ...]

Typical examples of phrases acceptable in the three clause categories are:

Introductory clause

(1) I would like ...
(2) Please find for me ...
(3) I have need of ...
(4) I desire ...

Document clause

(1) ... documents in the system ...
(2) ... information ...
(3) ... any available book or article in the repository ...
(4) ... references from the files ...
(5) ... all the stuff ...

Data clause

(1) ... written by Carr between 1958 and 1965.
(2) ... published in 1960 on information retrieval and work association but not programming.
(3) ... dated September 1966 by J.H. Smith, Joe Doe but not K.L. Jones about analog computers.

In the event that a word appearing in either the introductory or the document clause is not recognized, the computer initiates a man-machine dialogue in order to determine whether the word is essential and, if so, to seek out a synonym in its vocabulary. Examples of such dialogues appear on the attached typewriter printout.

```
YOU MAY PROCEED.:=     PLEAW-SE FIND FOR ME BOOKS
    CONCERNING STATISTICAL FUNCTIONS OR STANDARD
    DEVIATION, BUT NOT BUSINESS ORIENTED ZILKS
    INC---ENTITLED 'RUNCIBLE''I'.<>


PRINT?  :=     YES<>


YOUR MESSAGE IS:

01) PLEASE FIND FOR ME BOOKS
02) CONCERNING STATISTICAL FUNCTIONS OR STANDARD
03) DEVIATION, BUT NOT BUSINESS ORIENTED ZILKS
04) ENTITLED 'RUNCIBLE''I'.


CORRECTIONS?  :=     YES<>


LINE NO.  :=     3<>
LINE(S):

    DEVIATION, BUT NOT BUSINESS ORIENTED<>


MORE? :=     YES<>


LINE NO.  :=     0<>
LINE(S):

    DEAR COMPUTER:<>


MORE? :=     NO<>
PRINT?  :=     YES<>


YOUR MESSAGE IS:

01) DEAR COMPUTER:
02) PLEASE FIND FOR ME BOOKS
03) CONCERNING STATISTICAL FUNCTIONS OR STANDARD
04) DEVIATION, BUT NOT BUSINESS ORIENTED
05) ENTITLED 'RUNCIBLE''I'.


CORRECTIONS?  :=     NO<>

WE DID NOT RECOGNIZE THE WORD DEAR          .IS THIS WORD ESSENTIAL TO
    THE MEANING OF YOUR SENTENCE? :=     NO<>
```

```
WE DID NOT RECOGNIZE THE WORD COMPUTER     .IS THIS WORD ESSENTIAL TO
   THE MEANING OF YOUR SENTENCE? :=     NO<>

RETRIEVE          (  $B        (      (STATISTICAL FUNCTIONS
     )      +      (STANDARD     DEVIATION          )
     )      *      (BUSINESS     ORIENTED           )
     )      &      ( $A3  RUNCIBLE            I
     )

000001 'REFERENCES' HAVE BEEN RETRIEVED.

YOU MAY PROCEED.:=      PLEASE GET BOOKS BY--WRITTEN, EDITED
    OR PUBLISHED BY CARR.<>


PRINT?  :=     NO<>

RETRIEVE          (      ( $A1  CARR        + $A4  CARR        )
     *      ( $A5  CARR        )      )

000009 'REFERENCES' HAVE BEEN RETRIEVED.

PRINT SOME?  :=      NO<>

YOU MAY PROCEED.:=      GET BOOKS BY EITHER CARR OR RUBINOFF
    BUT NOT BY CARR<>


PRINT?  :=     NO<>

RETRIEVE          ( $A1        (       CARR        )      +      (
RUBINOFF          )      )      *      ( $A1  CARR
     )

000001 'REFERENCES' HAVE BEEN RETRIEVED.


YOU MAY PROCEED.:=      OBTAIN FOR ME BOOKS WRITTEN IN 1961 <>


PRINT?  :=     NO<>

RETRIEVE      $A2  1961

000127 'REFERENCES' HAVE BEEN RETRIEVED.

PRINT SOME?  :=      NO<>


YOU MAY PROCEED.:=      I WOULD LIKE YOU TO FIND BOOKS
    WRITTEN, EDITED, AND PUBLISHED BY CARR.<>


PRINT?  :=     NO<>

RETRIEVE          (      ( $A1  CARR        & $A4  CARR        )
     &      ( $A5  CARR        )      )

NO 'REFERENCES' HAVE BEEN RETRIEVED.
```

5.2 Real English

As mentioned above, each Easy English retrieval request consists of three clauses. Each clause contains one or more words or phrases to which a syntax class value has been assigned. A word's value determines the clause in which it may appear.

The program sequentially extracts words or phrases from the user's request, locates the word in the dictionary, finds its assigned value, and stores this value on an introduction code list. The process continues until a word or phrase is found whose value indicates that this word or phrase belongs to the document clause. At this point, introduction transformations are applied to the introduction code list to test for a valid introductory clause. Having found such a clause, the program repeats the process for the document clause with the exception that a word whose value indicates that it belongs to the data clause is the signal that the document code list is complete.

Figure 1 shows the basic processes which constitute the Real English system and the place that Real English has as an intermediary between user and machine as a translator of the user's request for information. The executive processor directly handles all conversation between the user and the machine, translating the user's request, expressed in English, into a form understandable by the information retrieval system, and communicating to the user, in

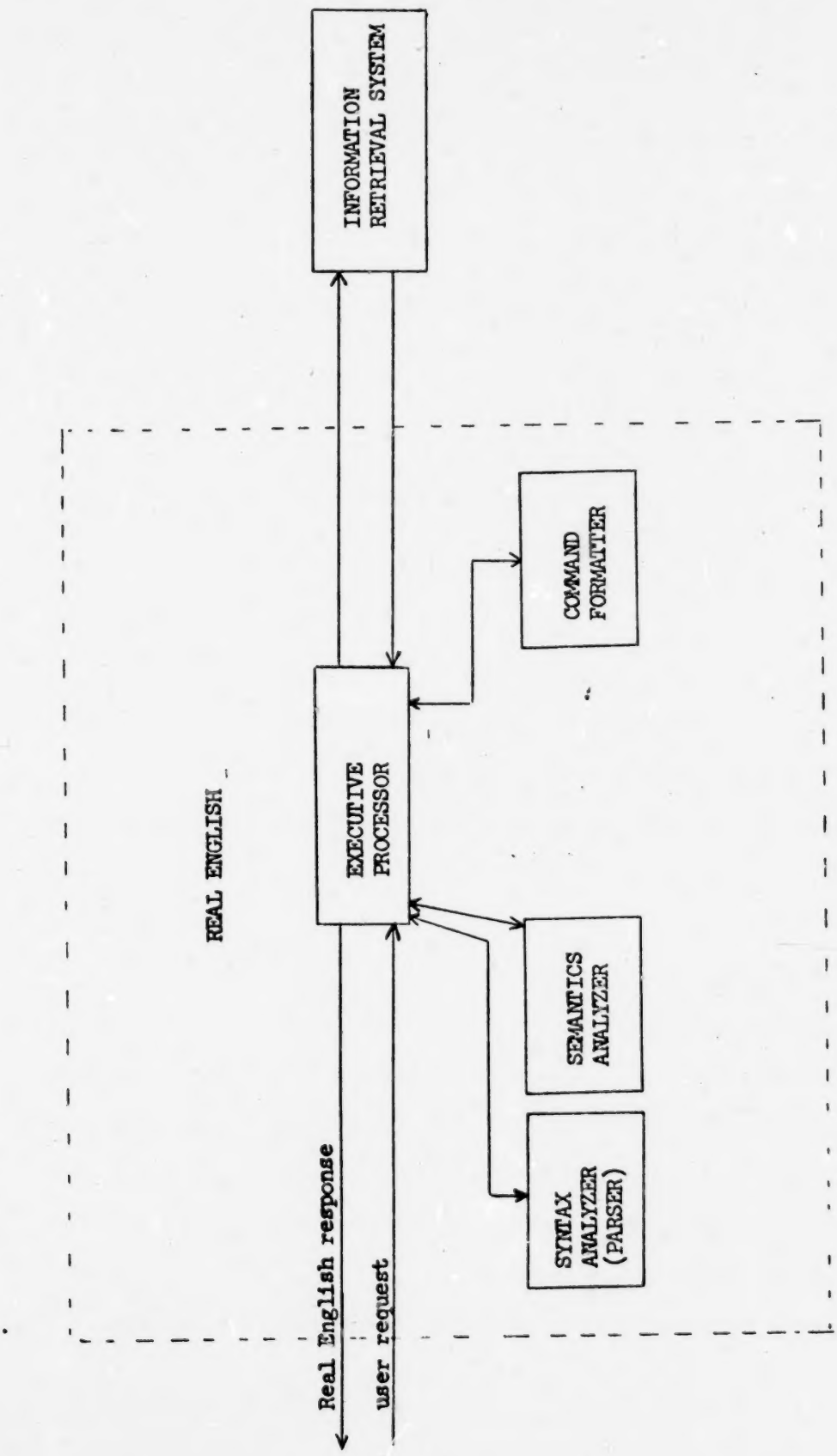

INFORMATION RETRIEVAL SYSTEM
REAL ENGLISH
COMMAND FORMATTER
EXECUTIVE PROCESSOR
SEMANTICS ANALYZER
SYNTAX ANALYZER (PARSER)
Real English response
user request

Figure 1

easily understandable form, the results of the search performed by the retrieval system if it is successful. In case of failure, either because Real English is unable to understand the user's request or because the system does not contain the information requested by the user, Real English informs the user of the cause of failure, and advises him as to how to proceed.

The normal flow of control within Real English is from syntax analyzer to semantic analyzer to command analyzer, each move proceeding under the control of the executive processor; the user's request is first analyzed syntactically (using a grammar of the "string analysis" type) to produce a parse tree, a graph of the syntactic relationships among the words which form the request. The parse tree is then processed by the semantic analyzer to determine what information the user wants; the result of the semantic analysis is a retrieval command(s) in a form intermediate between English and Symbolic Command Language (the language understood by the information retrieval system). Finally the output of the semantic analyzer is fed into the command formatter, which writes in Symbolic Command Language, a retrieval command which will cause the retrieval system to pick out of its information files that information which the user has requested.

In order to process retrieval requests presented to it in English, Real English must be able to translate all of the various types of expression which people are likely to use in formulating requests. A typical dialogue between a user and Real English might be the following:

user: GIVE ME THE TITLE OF ANYTHING BY JONES ON DDT.

system: AUTHOR=JONES

SUBJECT=DDT

TITLE=AGRICULTURAL USES OF DDT;

DDT AS AN INSECTICIDE

user: WHO PUBLISHED THESE BOOKS?

system: TITLE=AGRICULTURAL USES OF DDT

PUBLISHER=S.DAK.DEPT.OF AGRICULTURE

TITLE=DDT AS AN INSECTICIDE

PUBLISHER=SMITH AND CO.

user: I WOULD LIKE TO HAVE THE DATES OF ISSUE OF THESE ITEMS.

system: TITLE=AGRICULTURAL USES OF DDT

DATE OF ISSUE=1959

TITLE=DDT AS AN INSECTICIDE

DATE OF ISSUE=1960

user: WHAT HAS SMITH WRITTEN ON DDT?

system: AUTHOR=SMITH

SUBJECT=DDT

THE SYSTEM HAS NO INFORMATION ON THIS

user: HOW ABOUT ANDERSON?

system: AUTHOR=ANDERSON

SUBJECT=DDT

TITLE=A RE-EVALUATION OF DDT.

In this short dialogue there appear examples of the four distinct forms in which requests can be made:

1. Declarative sentences: I would like to have the dates of issue of these items.
2. Interrogative sentences: Who published these books? What has Smith written on DDT?
3. Imperative sentences: Give me the title of anything by Jones on DDT.
4. Sentence fragments: How about Anderson?

It is important to note that the fourth type of request, the sentence fragment, while not a well-formed English sentence, is very common in requests for information, and must therefore be recognized by Real English in order to maintain the user's ability to submit requests in the manner most usual to him.

Notice further that some of the requests (e.g., Give me the title of anything by Jones on DDT.; What has Smith written on DDT?) can be answered independently of their context in an extended man-machine dialogue; others (e.g., Who published these books?; I would like to have the dates of issue of these items.; How about Anderson?) depend upon the context in which they are used; thus: How about Anderson? is asking for one response if it immediately follows: What has Smith written on DDT? and is asking for a totally different response if it immediately follows: Is Jones the subject of a report on the medical profession? In the first case, the user wants to know first if Smith has written anything on DDT and then if Anderson has written anything on DDT. In the second case, the user wants to know if Jones is the subject of a report on the medical profession and then if Anderson is the subject of such a report. Requests whose meanings do not depend

upon the context of their use are called "contextually independent requests" while requests which do depend upon context are called "contextually dependent requests". Real English is able to handle both sorts of requests.

So far the operation of Real English under ideal conditions has been described; that is, we have, up to this point, seen what Real English does if it has succeeded in understanding the user's request. There are, however, a number of different cases in which Real English cannot produce symbolic retrieval commands from the user's request; these cases are of three types:

(1) Real English does not recognize one or more words in the user's request.

(2) Real English recognizes all of the words in the user's request, but cannot find a proper syntactic analysis for it.

(3) Real English can syntactically analyze the user's request, but cannot properly perform the semantic analysis necessary for translation into Symbolic Command Language.

There are two distinct meanings that "non-recognition of a word" can have in the context of the Real English system. In order to understand the distinctions between the two it is necessary to understand the operation of the syntax analyzer in some detail. The syntax analyzer, which forms a major part of Real English (Figure 2), consists of a parsing program, a dictionary, and a grammar; contained in the dictionary entry of a word are both syntactic information (e.g., syntactic category: noun, verb, adverb, etc.) and semantic information (e.g., the fact that the

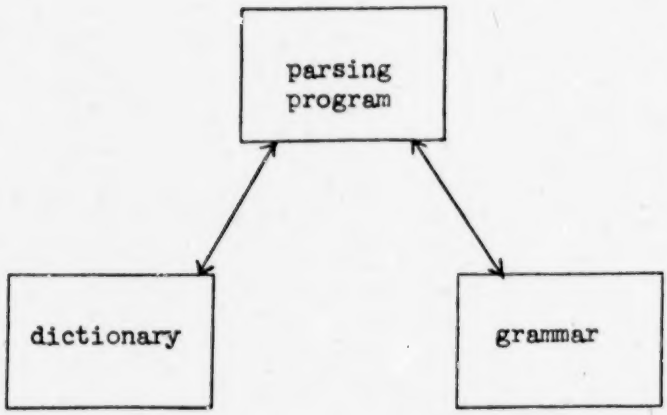


Figure 2: Syntax Analyzer

word "written" is indicative of a request for the name of the author of a document as in: What has Jones written on radar?).

It is important to note that the Real English dictionary contains only words in the "basic stock of English" and, in general, does not contain the technical terms such as DDT, radar, cardiac arrest, etc., which constitute the information stored in the files of a retrieval system; the reason for this will be explained presently.

In order to simplify matters, it is possible to conceive of the grammar as a list of sentential forms (strings of words of specific syntactic categories) which requests in English might take. The parsing program, in this extremely simplified model, tries to find each word of a request in the dictionary and then determines if the sequence of words forming the request constitutes a permissible sentential pattern. An example of a very simple sentential pattern with its associated parse tree (graph of the syntactic relationships among its words) is shown

in Figure 3. The words in the request which Real English would have in its dictionary would be "who", "has", "written" and "about". The word "radar", a technical term and therefore not in the dictionary, would be classified as an index term, a form of noun on which a retrieval is to be performed, (In this case, the system would look for all records in its information file whose subject is "radar".) and would be temporarily added to the dictionary under that classification. Since the sentential pattern:

(INTERROGATIVE PRONOUN)(AUXILIARY VERB)(PAST PARTICIPLE)(PREPOSITION)
(INDEX TERM)

is one recognized by the grammar, Real English would produce its parse tree and would then be able to go on to a semantic analysis whose end result would be the appropriate retrieval command in Symbolic Command Language. In the sense just described, we might say that Real English does not immediately recognize the word "radar" but is able, under the assumption that it is a technical term, (index term), to properly proceed with its analysis.

Another type of case in which Real English would be unable to "recognize" a word in a request is illustrated in the following examples:

WHOO HAS WRITTEN ABOUT RADAR?

WHO HAS BFXLQ ABOUT RADAR?

(Note that if the request were: Who has written about BFXLQ? the system would have no way of knowing that BFXLQ is not a proper technical term and would generate Symbolic Command Language commands

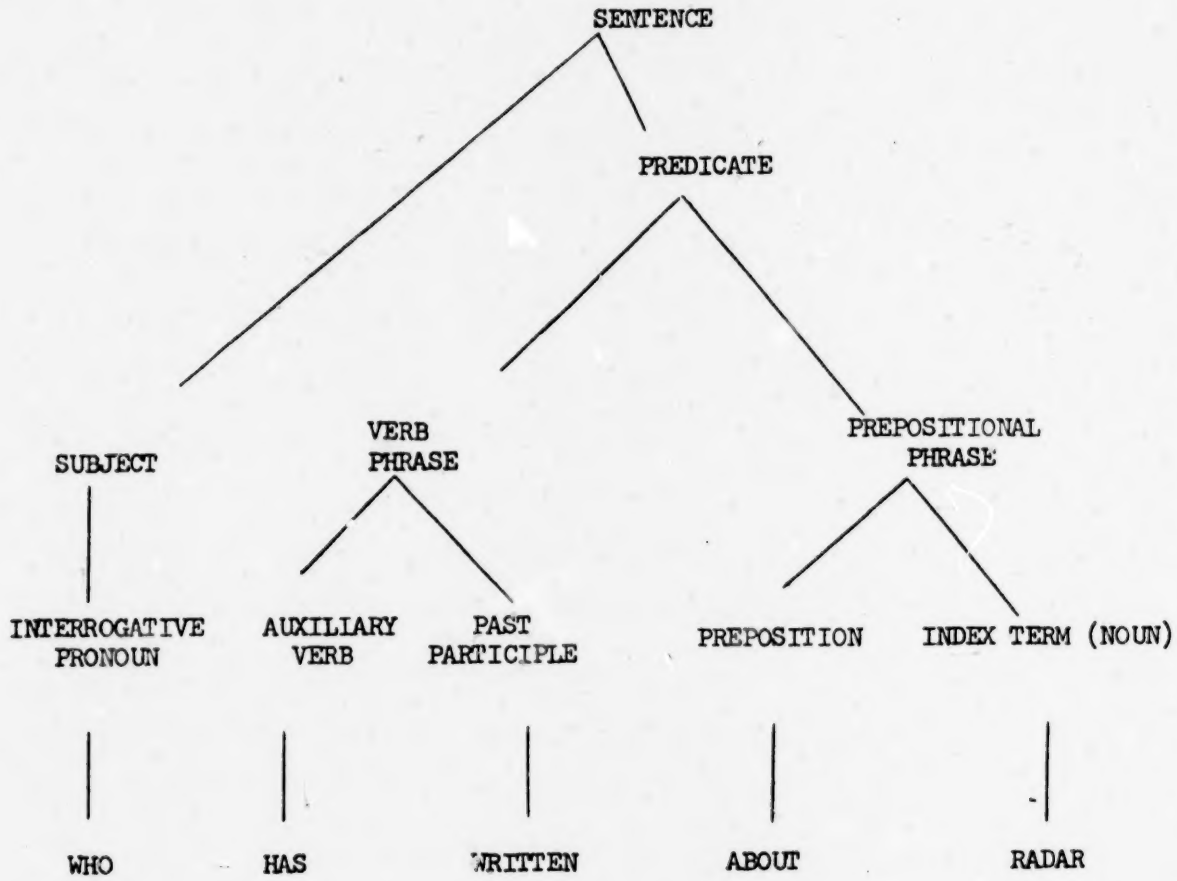


Figure 3

to retrieve all records whose subject is BFXLQ; there presumably being none, Real English would respond: SUBJECT=BFXLQ; THE SYSTEM HAS NO INFORMATION ON THIS., exactly as it would answer: SUBJECT=RADAR; THE SYSTEM HAS NO INFORMATION ON THIS., in case the request were: Who has written about radar? and the system had no information about radar.)

In the first case, the user has misspelled the word "Who"; and in the second, he has used what is presumably a nonsense word, "BFXLQ", in a position in which a past participle (e.g., "written")

would normally appear. This type of situation fits under the second sort of Real English failure, namely, failure to be able to properly syntactically analyze the request. As has been mentioned, if Real English cannot find a word which is part of a request in its dictionary, it assumes that the word is an index term and proceeds on that assumption. Thus, in the cases of the examples above, "WHOO" and "BFXLQ" would be added to the dictionary as index terms. Real English, in attempting to process either of the requests, would not find itself in trouble until it attempted to syntactically analyze the request and found that there are no English sentential forms which it recognizes which have index terms in the positions in which "WHOO", and "BFXLQ" appear. Since no syntactic analysis is possible, Real English must inform the user that his request cannot be properly translated and must, furthermore, indicate where it ran into trouble so that the user will be able to intelligently proceed to rephrase his request or go on to a different line of questioning. A closer examination of the second of the two improperly formed requests: Who has BFXLQ about radar?, shows that although it is not an English sentence (nor even a meaningful fragment), its first two words form a proper beginning of an English sentence. Thus, in attempting to analyze it syntactically, Real English would get as far as the parse tree of Figure 4. Any attempt to proceed further with the analysis would break down; thus, Real English would recognize the fact that the source of trouble is, in fact, the word "BFXLQ", the first word in the request past which syntactic analysis is impossible.

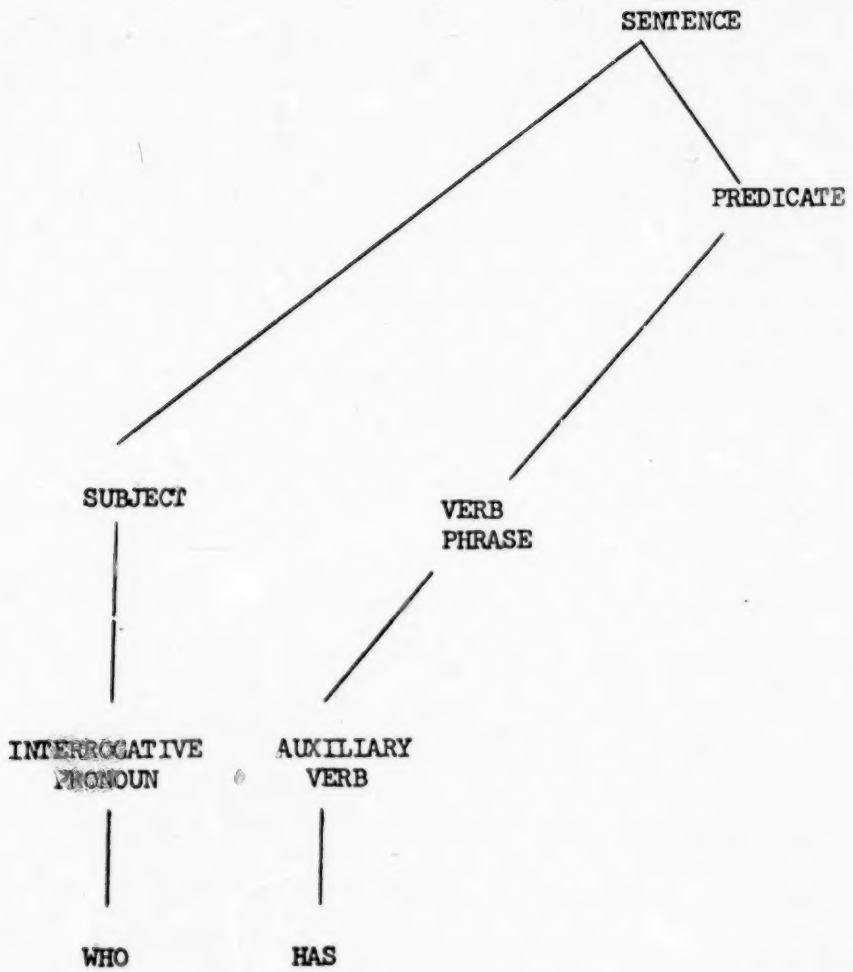

SENTENCE
PREDICATE
SUBJECT
VERB
PHRASE
INTERROGATIVE
PRONOUN
AUXILIARY
VERB
WHO
HAS

Figure 4

Real English would then inform the user of the problem by issuing the response:

> THE SYSTEM DOES NOT RECOGNIZE THE WORD BFXLQ IN THE WAY THAT YOU HAVE USED IT. PLEASE REFORMULATE YOUR REQUEST.

In the case of the improperly formed request: Whoo has written about radar?, the syntactic analysis would never get past the beginning of the sentence, and so Real English would issue the response:

> THE SYSTEM DOES NOT RECOGNIZE THE WORD "WHOO" IN THE WAY THAT YOU HAVE USED IT. PLEASE REFORMULATE YOUR REQUEST.

Note that a similar sort of response is appropriate even in the case of a request all of whose words Real English recognizes, but which is nevertheless improperly formed, e.g., a request of the sort:

> WHO IS IT THE AUTHOR OF THIS BOOK?

In this case, Real English would get as far as the word "the" in its syntactic analysis; here, however, it would be inappropriate for the system to tell the user that it does not recognize the word "the" as he has used it. Rather, the system issues the response:

> THE SYSTEM CANNOT UNDERSTAND THE WAY YOU HAVE FORMULATED YOUR REQUEST; THE TROUBLE IS IN THE VICINITY OF THE PHRASE UNDERLINED:
>
> WHO IS IT THE AUTHOR OF THIS BOOK?
>
> PLEASE REFORMULATE YOUR REQUEST.

Notice that in both cases, that of words not recognized and that of (syntactically) improperly formed requests, Real English is able to

locate the source of trouble (and communicate it to the user) by keeping track of how far into the request syntactic analysis is possible. In the case of a request in which more than one word is not recognized, more than one faulty use of recognized words is encountered or a combination of both unrecognized words and faulty constructions occur, Real English is able to locate one error at a time. Thus, the user might ask:

WHO IS IT THE BFXLQ OF THIS BOOK?

in which case the system would return with:

THE SYSTEM CANNOT UNDERSTAND THE WAY YOU HAVE FORMULATED YOUR REQUEST; THE TROUBLE IS IN THE VICINITY OF THE PHRASE UNDERLINED:

WHO IS IT THE BFXLQ OF THIS BOOK?

PLEASE REFORMULATE YOUR REQUEST.

If the user then reentered the request as:

WHO IS THE BFXLQ OF THIS BOOK?

the system would return with:

THE SYSTEM DOES NOT RECOGNIZE THE WORD "BFXLQ" IN THE WAY YOU HAVE USED IT. PLEASE REFORMULATE YOUR REQUEST.

On the other hand, if the user were to respond with:

WHO IS THE AUTHOR OF THIS BOOK?

immediately after the system's first error message, his request would be properly processed and would result in the system's printing

out the name of the author of the book whose information record it had previously found. (Assuming that a previous user request had resulted in the retrieval of such a record.)

The third sort of case in which Real English must initiate further dialogue with the user before it is able to properly fill his request is described above as semantic failure. The term "semantic failure" is meant to describe the situation in which Real English has succeeded in syntactically analyzing the user's request, but is incapable of producing symbolic retrieval commands from the semantic information contained in the request. Semantic failure itself falls into one of the three classes:

(1) The user's input is not a request for information of the sort contained in the information file.

(2) The user's request does not specifically enough indicate what information he wants.

(3) The user's input is a contextually dependent request not preceded by a contextually independent request.

In order to understand the case of a user input which is not a request for information of the sort contained in the information files, we must realize that Real English is designed to operate on a data base of information pertaining to a specific discipline, e.g. chemistry, physics, bibliography, psychology, etc. In what follows, we shall assume the data base of the Toxicology Information File project, a file whose words contain toxicological information about chemical compounds. A request like: What are the effects of prolonged exposure to the sun?, although it does not request information of a toxicological

nature would, nevertheless, be processed by Real English into a symbolic command to retrieve the effects of prolonged exposure to the sun because the sentential form of the request is one recognized by the system as the form of a possible toxicological request (e.g., What are the effects of prolonged exposure to DDT?). Since the data base contains no information on the effects of exposure to the sun, Real English would answer the user: THE SYSTEM HAS NO INFORMATION ON THIS. If, on the other hand, the user typed one of the following into the system:

1. WHO WON THE 1950 WORD SERIES?
2. WHY IS THE SKY BLUE?

or even

3. LET'S RETURN TO MY PREVIOUS LINE OF QUESTIONING.

Real English would not recognize the input as having the sentential form of a possible toxicological request. In this case, the system would answer the user:

> THE SYSTEM HAS NOT RECOGNIZED YOUR INPUT AS A REQUEST FOR TOXICOLOGICAL INFORMATION. THE SYSTEM IS READY TO ACCEPT A REQUEST.

Note that although sentence 3 above might possibly be part of a dialogue in which a person is attempting to elicit information from either another person or from a machine, Real English is not equipped to process it. First, it does not have the form of a request for specific information and second the ability to process it would require Real English to keep track of all previous dialogue and to be able to return to any segment of previous dialogue, facilities

which would enormously complicate the task which Real English has to perform.

The second sort of semantic failure, lack of specificity in the user's request, is failure only in the sense that further dialogue is necessary before Real English can generate the proper retrieval commands. An example of this sort of request is: What are the effects of DDT? The following is a list of the qualified category headings of information fields regarding the effects of a chemical substance that a user might want:

ANIMAL EXPERIMENTS. ORAL ADMINISTRATION. EFFECT

ANIMAL EXPERIMENTS. DERMAL ADMINISTRATION. EFFECT

ANIMAL EXPERIMENTS. INJECTION. EFFECT

ANIMAL EXPERIMENTS. SKIN APPLICATION. EFFECT

ANIMAL EXPERIMENTS. EYE APPLICATION. EFFECT

CLINICAL EFFECTS. HUMAN. ABSORPTION. ORAL. ACUTE. EFFECT

CLINICAL EFFECTS. HUMAN. ABSORPTION. ORAL. SUBACUTE. EFFECT

CLINICAL EFFECTS. HUMAN. ABSORPTION. ORAL. CHRONIC. EFFECT

CLINICAL EFFECTS. HUMAN. DERMAL. ACUTE. EFFECT

CLINICAL EFFECTS. HUMAN. DERMAL. SUBACUTE. EFFECT

CLINICAL EFFECTS. HUMAN. DERMAL. CHRONIC. EFFECT

CLINICAL EFFECTS. HUMAN. INHALATION. ACUTE. EFFECT

CLINICAL EFFECTS. HUMAN. INHALATION. SUBACUTE. EFFECT

CLINICAL EFFECTS. HUMAN. INHALATION. CHRONIC. EFFECT

LOCAL EFFECT. DERMAL. ACUTE. EFFECT

LOCAL EFFECT. DERMAL. SUBACUTE. EFFECT

LOCAL EFFECT. DERMAL CHRONIC. EFFECT

LOCAL EFFECT. EYE APPLICATION. ACUTE. EFFECT

LOCAL EFFECT. EYE APPLICATION. SUBACUTE. EFFECT

LOCAL EFFECT. EYE APPLICATION. CHRONIC. EFFECT

AFTEREFFECT. ONSET

AFTEREFFECT. DURATION

AFTEREFFECT. RECURRENCE

AFTEREFFECTS OTHER THAN CASE HISTORY. ACCOUNT SUBCLINICAL EFFECTS. EFFECT

Unless the user actually wanted all available information on DDT (which the system should certainly be able to provide if this is indeed the case), he would probably be swamped by a massive printout of the information contained under all of the above headings. (Especially considering the relatively slow output speed of the teletype terminal and the relatively restricted amount of simultaneous output possible on the video display terminal.) In such a case, Real English initiates a dialogue with the user informing him of the sorts of information available given his initial request and asking him to further qualify the request (if he is not interested in all available information) before a retrieval and/or printout is performed.

The third sort of semantic failure, the use of an input in the form of a contextually dependent request not preceded by a contextually independent request, also results in a sense from a lack of specificity; thus, the request: How about Anderson? in isolation from an immediately preceding request in which a name appears (e.g., Has Jones written about DDT?) cannot be processed since it has no interpretable meaning as a request for information. In such a case, the system responds:

YOU HAVE NOT PROVIDED ENOUGH INFORMATION TO SPECIFY
EXACTLY WHAT YOU WANT. PLEASE REPHRASE YOUR REQUEST
GIVING MORE DETAIL.

By recognizing properly phrased English sentences, regardless of how colloquial the English may be, Real English provides the user of a typewriter console with the ability to search a computerized file of data without any knowledge of computers, typewriter consoles, or even how the system works. By calling ambiguities to his attention, Real English helps the user to get maximum information retrieval with minimum concern for possible errors in his own portion of the man-machine dialogue. For the occasional user of the system, Real English means immediate access to the files with little or no direction from an operator and no prior training whatsoever.

BIBLIOGRAPHY

6.1 Published Papers

1. Rubinoff, M. and White, J. F., Jr.; Establishment of the ACM Repository and the Principles of the IR System Applied to its Operation. Comm. ACM, 8:595, 1965.

2. Rubinoff, M.; A Look Ahead. Proceedings of the Second National Symposium on Engineering Education. Engineers Joint Council, October 1965.

3. Rubinoff, M.; A Rapid Procedure for Launching a Microthesaurus. IEEE Trans. Engl. Writing and Speech, August 1966.

4. Rubinoff, M.; Why Sigir? ACM SIGIR FORUM, Vol. IV, No 2, 1967.

5. Rubinoff, M., Bergman, S., Cautin, H., and Rapp, F.; Easy English, A Language for Information Retrieval Through a Remote Typewriter Console. Comm. ACM, 10: 693, 1968.

6. Lowe, T. C.; Retrieval from Direct-Access Memory Using Truncated Record Names, Software Age, August 1967.

7. Rubinoff, M. and Stone, D. C.; Semantic Tools in Information Retrieval, Proceedings of the American Documentation Conference, June 1967.

8. Rubinoff, M., Bergman, S., Franks, W., and Rubinoff, E.,; Experimental Evaluation of Information Retrieval Through a Teletypewriter, Comm. ACM, 9:598, 1968.

9. Rubinoff, M.; "Education in Computer Engineering," Journal of Engineering Education, April 1968.

10. Rubinoff, M; "Information Retrieval and Systems Engineering," Proceedings of the First World Congress of Engineers and Architects in Israel, December 1967.

11. Lowe, T. C.; An article on "Triplet Searching" entitled "Encoding from Alphanumeric Names to Record Addressis," Software Age, April 1968.

12. Stone, D. C. and Rubinoff, M.; "Statistical Generation of a Technical Vocabulary," American Documentation, October 1968.

6.2 Internal Reports

1. Rubinoff, M. and White, J. F., Jr.; Description of Cataloging and Indexing System for the ACM Repository. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphea. 1965.

2. Rubinoff, m., et al; The Moore School Information Systems Laboratory. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. June 1966.

3. Fischer, Stephen B.,; An Executive Control System for Information Retrieval via a Remote Console. M.S. Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. 1966.

4. Rubinoff, M., et al; Summary Description of Easy English. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. February 1967.

5. Cautin, H. and Rapp, F.; Description of Easy English. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. April 1967.

6. Fogel, M.; On-Line Typewriter Access to Classification Tables on Drum Storage. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. April 1967.

7. Lowe, T. C.; Retrieval from Direct-Access Memory Using Truncated Record Names. The Moore School of Electrical Engineering, University of Pennsylvanis, Philadelphea. May 1967.

8. Cautin, H., Lowe, T. C., Rapp, F., Rubinoff, M.; An Experimental On-Line Information Retrieval System. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. May 1967.

9. Rubinoff, M. and Stone, D. C.; Semantic Tools in Information Retrieval. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. May 1967.

10. Rubinoff, M., Franks, W., and Stone, D. C.; Description of an Experiment Investigating Term Relationships as Interpreted by Humans. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. June 1967.

11. Smith, J. M.; An Oral Experiment on Retrieval Dialogue. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. June 1967.

12. Smith, J. M.; A Written Experiment on Retrieval Dialogue. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. August 1967.

13. Edwards, J. S.; Adaptive Man-Machine Interaction in Information Retrieval. Ph.D. Dissertation presented to the Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. December, 1967.

14. Stone, D. C.; Word Statistics in the Generation of Semantic Tools for Information Systems. Master's Thesis presented to the Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. December 1967.

15. Crowley, John Donegan, Jr.; Design and Implementation of a Large Scale File Structure for an On-Line Indexing/Retrieval System. Master's Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvanis, Philadelphia. December 1968.

16. Libove, George Alan; Automatic Generation of Synonyms. Master's Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. May 1969.

17. Cimprich, Jack Robert; Programming Considerations in the Implementation of an English Language Recognizer. Master's Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. May 1969.

18. Fogel, Marc; Determination of Statistical Clumps. Master's Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. May 1969.

19. Cautin, Harvey; Real English: A Translator to Enable Natural Language Man-Machine Conversation. Ph.D. Dissertation presented to The Moore School of Electrical Engineering, University of Pennsylvania. May 1969.

20. Klappholz, David A.; Real English - A Description of Its Operation. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. March 1970.

21. The Moore School Information Systems Laboratory; SOLER - System for On-Line Entry and Retrieval. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. March 1971.

22. Hirschfeld, Leonard Jay; Design and Implementation of the Retrieval Mechanism of the SOLER Storage and Retrieval System. M.S. Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. August 1971.

23. Carlson, Clifford Hugh; Update Phase of SOLER. M.S. Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. August 1971.

24. Kaplan, Gerald; Design and Implementation of the Invert Phase of a Multiple Data Base Information Retrieval System. M.S. Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. August 1971.

6.3 References on Information Retrieval

1. Baker, Frank B. (1965), "Latent Class Analysis as an Association Model for Information Retrieval," in Statistical Association Methods for Mechanized Documentation, edited by M. E. Stevens, Vincent E. Giuliano, and Laurence B. Heilprin, National Bureau of Standards Miscellaneous Publication 269.

2. Borko, Harold, and Myrna Bernick (1963), "Automatic Document Classification," Journal of the Association for Computing Machinery, 10:151-162.

3. Borko, Harold, and Myrna Bernick (1964), "Automatic Document Classification; Part II; Additional Experiments," Journal of the A.C.M., 11:138-151.

4. Cleverdon, Cyril W., Jack Mills, and Michael Keen (1966), Aslib-Cranfield Research Project; Factors Determining the Performance of Indexing Systems; vol. 1 (2 parts) and vol. 2.

5. Curtice, Robert M., and Paul E. Jones (1967), "Distributional Constraints and the Automatic Selection of an Indexing Vocabulary," Proceedings of the American Documentation Institute, vol. 4.

6. Damerau, Fred J. (1965), "An Experiment in Automatic Indexing," American Documentation, 16:283-289.

7. Dennis, Sally F. (1965), "The Construction of a Thrsaurus Automatically from a Sample of Text," in Statistical Association Methods for Mechanized Documentation, ed. by M. D. Stevens, et al., N.B.S. Misc. Publication 269.

8. Dennis, Sally F. (1967), "The Design and Testing of a Fully Automatic Indexing-Searching System for Documents Consisting of Expository Text," in Information Retrieval: A Critical View, ed. by George Schecter.

9. Doyle, Lauren B. (1961), "Semantic Road Maps for Literature Searchers," Journal of the A.C.M., 8:553-578.

10. Doyle, Lauren B. (1965), "Expanding the Editing Function in Language Data Processing," Communications of the A.C.M., 8:238-243.

11. Edmundson, H.P., and R.E. Wyllys (1961), "Automatic Abstracting and Indexing--Sruvey and Recommendations," Communications of the A.C.M., 4:226-234.

12. Edwards, John S. (1967), Adaptive Man-Machine Interaction in Information Retrieval, unpublished Ph.D. dissertation, The Moore School of Electrical Engineering, University of Pennsylvania.

13. Giuliano, Vincent E. (1965), "The Interpretation of Word Associations," in Statistical Association Methods for Mechanized Documentation, ed. by M. E. Stevens, et al., N.B.S. Misc. Publ. 269.

14. Giuliano, Vincent E., and Paul E. Jones (1963), "Linear Associative Information Retrieval," in Vistas in Information Handling, vol. I, ed. by Paul W. Howerton.

15. Giuliano, Vincent E., and Paul E. Jones (1966), Study and Test of a Methodology for Laboratory Evaluation of Message Retrieval Systems, Interim Report ESD-TR-66-405, Decision Sciences Lab., L. G. Hanscom Field, U. S. Air Force, Bedford, Mass.

16. Haibt, Luther, Margaret Fischer, Robert Ketelhut, and Jay Ogg (1967), "Finding 4000 References without Indexing" (An Effectiveness Study of Full Text Searching), presented at The Fourth Annual National Colloquium on Information Retrieval, May, 1967, Philadelphia, Pa.

17. Henderson, Madeline, John Moats, Mary Stevens, and Simon Newman (1966), Cooperation, Convertibility, and Compatibility Among Information Systems: A Literature Review, National Bureau of Standards Miscellaneous Publication 276. See especially Section 3.7, Systematization and Terminology Control.

18. Herner, Saul (1963), "The Role of Thesauri in the Convergence of Word and Concept Indexing," in Automation and Scientific Communication, Short Papers, 26th Annual Meeting, American Documentation Institute, edited by H. P. Luhn.

19. IFIP-ICC Vocabulary of Information Processing (1966), First English Language Edition, North-Holland Publishing Co., Amsterdam.

20. Jones, Paul E., and Robert M. Curtice (1967), "A Framework for Comparing Term Association Measures," American Documentation, 18:153-161.

21. Kuhns, J. L. (1965), "The Continuum of Coefficients of Association," in Statistical Association Methods for Mechanized Documentation, ed. by M. E. Stevens, et al., N.B.S. Misc. Publication 269.

22. Lewis, P. A. W., P. B. Baxendale, and J. L. Bennett (1967), "Statistical Discrimination of the Synonymy/Antonymy Relationship between Words," Journal of the A.C.M., 14:20-44.

23. Luhn, H. P. (1958), "The Automatic Creation of Literature Abstracts," I.B.M. Journal of Research and Development, 2:159-165.

24. Maron, Melvin E. (1961), "Automatic Indexing: An Experimental Inquiry," Journal of the A.C.M., 8:404-417.

25. Maron, Melvin E., and J. L. Kuhns (1960), "On Relevance, Probabilistic Indexing and Information Retrieval," Journal of the A.C.M., 7:216-244.

26. Miller, G. A., E. B. Newman, and E. A. Friedman (1958), "Length-Frequency Statistics for Written English," Information and Control, 1:370-389.

27. Needham, R. M. (1962), "A Method for Using Computers in Information Classification," Information Processing 1962, Proceedings of IFIP Congress 62, ed. by Cicely M. Popplewell, 1963.

27. O'Connor John (1965), "Automatic Subject Recognition in Scientific Papers: An Empirical Study," Journal of the A.C.M., 12:490-515.

28. Reisner, Phyllis (1965), "Semantic Diversity and a 'Growing' Man-Machine Thesaurus," in Some Problems in Information Science, ed. by Manfred Kochen.

29. Rubinoff, Morris, and Don C. Stone (1967), "Semantic Tools in Information Retrieval," Proceedings of the American Documentation Institute, Annual Meeting, vol. 4.

30. Salisbury, Blinn a., Jr., and H. Edmund Stiles (1967), "The Use of the B-Coefficient in Information Retrieval," Working Paper, R45, 67-12.

31. Salton, Gerard (1965), "Progress in Automatic Information Retrieval," I.E.E.E. Spectrum, 2:90-103.

32. Salton, Gerard (1966), "Information Dissemination and Automatic Information Systems," Proceedings of the I.E.E.E., 54:1663-1678.

33. Stiles, H. Edmund (1961), "The Association Factor in Information Retrieval," Journal of the A.C.M., 8:271-279.

34. Walston, Claude E. (1965), "Information Retrieval," in Advances in Computers, vol. 6, ed. by Franz L. Alt and Morris Rubinoff, Academic Press, New York.

35. Williams, J. H. (1965), "Results of Classifying Documents with Multiple Discriminant Functions," in Statistical Association Methods for Mechanized Documentation, ed. by M. E. Stevens, et al., N.B.S. Misc. Publication 269.

36. Winters, William K. (1965), "A Modified Method of Latent Class Analysis for File Organization in Information Retrieval," Journal of the A.C.M., 12:356-363.

6.4 References on Statistical Clumping

1. Dale, A. G. and Dale, N; "Clumping Techniques and Associative Retrieval," NBS Miscellaneous Publication, No. 269, U.S. Government Printing Office, 1965.

2. Edwards, John S.; Adaptive Man-Machine Interaction in Information Retrieval, Ph.D. Dissertation presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. December 1967.

3. Haigler, Sandra L.; A Program to Generate A Concordance From Text, M.S. Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. December 1967.

4. Jones, Karen Spark and Jackson, David; "Current Approaches To Classification and Clumpfinding at the Cambridge Language Research Unit," Computer Journal, Vol. 10, No. 1, p. 29, May 1967.

5. Jones, Paul E. and Curtice, Robert M.; "A Framework for Comparing Term Association Measures," American Documentation, July 1967, p. 153.

6. Needham, R. M. (1963); "A Method for Using Computers In Information Classification," Information Processing 62: Proceedings of IFIP Congress 1962, Amsterdam: North Holland Publishing Co., p. 284.

7. Salisbury, Blinn A., Jr. and Stiles, H. Edmund; "The Use of The B-Coefficient in Information Retrieval."

6.5 References on Computerized English

1. Codd, George G.: "Elements of Data Management Systems," Computing Surveys, Vol. 1, No. 2, June 1969, pp. 117-133.

2. Hirschfeld, Leonard Jay: Design and Implementation of the Retrieval Mechanism of the SOLER Storage and Retrieval System. Master's Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. August 1971.

3. Kaplan, Gerald: Design and Implementation of the Invert Phase of a Multiple Data Base Information Retrieval System. Master's Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. August 1971.

4. Carlson, Clifford Hugh: Update Phase of SOLER. Master's Thesis presented to The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. August 1971.

5. Cautin, Harvey: Real English: A Translator to Enable Natural Language Man-Machine Conversation. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. 1969.

6. Harris, Z.: String Analysis of Sentence Structure, Mouton and Co., The Hague, 1962.

7. Joshi, A., Kosaraju, S., and Yamada, H.: String Adjunct Grammars. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. 1968.

8. Chomsky, N.: Syntactic Structures, Mouton and Co., The Hague, 1967.

9. _____: Aspects of the Theory of Syntax, M.I.T. Press, Cambridge, Massachusetts. 1965.

10. Sager, N.: "Syntactic Analysis of Natural Language," Advances in Computers, Vol. 8, pp. 153-188, 1967.

11. _____: A Computer String Grammar of English, New York University Linguistic String Project, Report 3, New York, November 1968.

12. Felsen, J.: Documentation of the Implementation of Real English (unpublished report). The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. 1969.

13. Cimprich, J.: Programming Considerations in the Implementation of an English Language Recognizer. The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. 1969.

6.6 References on Automatic Indexing

1. Abraham, C. T.: "Techniques for Thesaurus Organization and Evaluation," Proceedings - American Documentation Institute, Vol. 1:485, Spartan Books, 1964.

2. Baker, F. T., et al.: Research on Automatic Classification, Indexing, and Extracting, Contract NONR 4456 (00) AD No. 485188, April, 1966.

3. Bar Hillel, Y.: "A Logician's Reaction to Theorizing on Information Retrieval," American Documentation, Vol. 3:103, 1957.

4. Bar Hillel, Y.: "Is Information Retrieval Approaching a Crisis?" American Documentation, Vol. 14:95, 1963.

5. Bar Hillel, Y.: Language and Information, Selected Essays on Their Theory and Application, Addison-Wesley, Reading, Mass., 1964.

6. Bobrow, D. G.: Problems in Natural Language Communication with Computers, Contract No. AF19(628)-5065, AD No. 639323, 1966.

7. Berul, L.: Information Storage and Retrieval a State-of-the-Art Report, Auerbach Corp., AD No. 630089, 1964.

8. Bonner, R. E.: "On Some Clustering Techniques," IBM J., p. 22, January, 1964.

9. Borko, and Bernick: "Automatic Document Classification Part I," J. ACM, Vol. 10, p. 151, 1963.

10. Borko and Bernick: "Automatic Document Classification Part II Results," J. ACM, Vol. 11, p. 138, 1964.

11. Bryant, E. C., et al.: Some Aspects of the Improvement of Document Screening, Contract No. AF49(638)-1484, AD No. 628191, 1965.

12. Cleverdon, C., et al.: ASLIB Cranfield Research Project - Factors Determining the Performance of Indexing Systems Volume 1. Design. Cranfield, England, 1966.

13. Cleverdon, C., and Keen, M.: ASLIB Cranfield Research Project - Factors Determining the Performance of Indexing Systems. Volume 2. Test Results. Cranfield, England, 1966.

14. Dale, A. G., and Dale, N.: "Some Clumping Experiments for Associative Document Retrieval," American Documentation, Vol. 16:5, 1965.

15. Dennis, S. F.: The Construction of a Thesaurus Automatically From a Sample of Text. Statistical Association Methods for Mechanized Documentation, NBS Misc. Publication 269, Washington, D. C.

16. Edmundson, H. P.: Mathematical Models of Synonymy, SDC, SP-1975/00/01, 1966.

17. Eveleigh, V. W.: Adaptive Control Systems, Electro-Technology, Vol. 71:78.

18. Feldman, J. A.: Aspects of Associative Processing, Contract No. AF19(628)-500, AD No. 614634, 1965.

19. Giuliano, V. E.: Analogue Networks for Word Association, IEEE Trans. on Mil. Electronics, Vol. MIL-7, No. 2 and 3, p. 221, 1963.

20. Giuliano, V. E.: The Interpretation of Word Associations, Statistical Association Methods for Mechanized Documentation, NBS Misc. Publication 269, Washington, D. C., 1965.

21. [illegible]liano and Jones: Study and Test of a Methodology for [illegible]oratory Evaluation of Message Retrieval Systems, [illegible]erim Report ESD-TR-66-405, Decision Science Lab., [illegible]scomb Field (USAF), Bedford, Mass., August, 1966.

22. [illegible], S.: On the Mechanical Simulation of Habit-Forming [illegible] Learning, J. Informat. Contr., Vol. 2, p. 226, September 1959.

23. Haibt, L., Fischer, M., et al.: An Effectiveness Study of Full Text Searching, A paper presented at the Fourth Annual National Colloquium on Information Retrieval, Philadelphia, Pennsylvania. May 1967.

24. Ivie, E. L.: Search Procedures Based on Measures of Relatedness Between Documents, Ph.D. Dissertation, MIT (Project MAC, MAC-TR-29(Thesis), 1966).

25. Jones, P. E., Curtice, R. M.: A Framework for Comparing Term Association Measures, American Documentation, Vol. 4:153, 1967.

26. Jones, P. E.: Historical Foundations of Research on Statistical Association Techniques for Mechanized Documentation, Statistical Association Methods for Mechanized Documentation, NBS Misc. Publication 269, Washington, D. C., 1965.

27. Kessler, M. M.: Comparison of the Results of Bibliographic Coupling and Analytic Subject Indexing, American Documentation, Vol. 16, No. 3. 1965.

28. Knable, J. P.: An Experiment Comparing Key Words Found in Indexes and Abstracts Prepared by Humans with Those in Titles, American Documentation, Vol. 3, No. 4, p. 123, 1965.

29. Knowlton, K. C.: A Programmer's Description of LLLLLL, Bell Telephone Laboratories' Low-Level List Language, Unpublished Bell Telephone Laboratories Technical Memorandum, MM 65-1271-2, February 1966.

30. Lamson, B. G., and Dimsdale, B.: A Natural Language Information Retrieval System, Proc, IEEE, Vol. 54, No. 12:1636, Dec., 1966.

31. Lewis, P. A. W., Baxendale, P. B., and Bennett, J. L.: Statistical Discrimination of the Synonymy/Antonymy Relationship Between Words, J. ACM, Vol. 14, No. 1, p. 20, 1967.

32. Luhn, H. P.: A Statistical Approach to Mechanized Encoding and Searching of Literary Information, IBM J., p. 309, 1957.

33. Luhn, H. P.: The Automatic Creation of Literature Abstracts, IBM J., p. 159, April 1958.

34. Lynch, M. F.: Computers in the Library, Nature, Vol. 212, p. 1402, 1966.

35. MuCutchen, C. W.: Random Code Numbers for Universal Identification of Documents, American Documentation, Vol. 16, No. 2, p. 91, 1965.

36. McMahon, L. E.: FASE: A Fundamentally Analyzable Simplified English. Bell Telephone Laboratories Technical Memorandum, MM 65-1221-7, 1965.

37. Maron, M. E. and Kuhns, J. L.: On Relevance, Probabilistic Indexing and Information Retrieval, J. ACM, Vol. 7, p. 216, 1960.

38. Maron, M. E.: Automatic Indexing: An Experimental Enquiry, J. ACM, 8:404, 1961.

39. Montague, B. A.: Testing, Comparison, and Evaluation of Recall, Relevance, and Cost of Coordinate Indexing with Links and Roles, American Documentation, Vol. 16, No. 3, p. 201, 1965.

40. Needham, R. M. and Sparck Jones, K.: Keywords and Clumps, J. of Doc., Vol. 20, No. 1, p. 5, 1964.

41. Nolan, J. F., Armenti, A. W.: An Experimental On-Line Data Storage and Retrieval System, Mass. Inst. of Tech., Tech. Rpt. No. 377, Lexington, Mass., 1965.

42. O'Connor, J.: Mechanized Indexing Methods and Their Testing, J. ACM, Vol. 11, No. 4, P. 437, 1964.

43. Prywes, N. S.: Browsing in an Automated Library Through Remote Access, University of Pennsylvania, The Moore School of Electrical Engineering, Philadelphia, Pa., Internal publication.

44. Rees, A. M.: The Aslib-Cranfield Test of the Western Reserve University Indexing System for Metallurgical Literature: A Review of the Final Report, American Documentation, Vol. 16, No. 2, p. 73, 1965.

45. Rocchio, J. J.: Information Storage and Retrieval. Rpt. No. ISR 10, Harvard Computation Laboratory, 1965 (Ph.D. Dissertation).

46. Ross, I. C.: Some Text Analysis Routines. Unpublished Bell Telephone Technical Memorandum MM-66-1221, August 1966.

47. Rubinoff, M. and White, J. F., Jr.: Establishment of the ACM Repository and Principles of the IR System Applied to Its Operation, Communication of the ACM, Vol. 8, No. 10, p. 595, 1965.

48. Sage, C. R., et al: Adaptive Information Dissemination, American Documentation, Vol. 16, No, 3, p. 185, 1965.

49. Salton, G.: The Evaluation of Automatic Retrieval Procedures - Selected Test Results Using the SMART System, American Documentation, Vol. 16, No. 3, p. 209, 1965.

50. Salton, G.: Information Dissemination and Automatic Information Systems, Proc. of the IEEE, Vol. 54, No. 12, p. 1663, 1966.

51. Shaw, R. R.: $F^{1}a^{1}c^{1}e^{2}\ I^{2}t^{3}s^{1}\ N^{3}o^{1}r^{2}m^{1}$, American Documentation, Vol. 16, No. 2, p. 77, 1965.

52. Simmons, R. F.: Natural Language Processing and the Time-Shared Computer. SDC Paper SP-1974/001/00 System Development Corp., Samta Monica, Calif., 1965.

53. Stiles, H. E.: The Association Factor in Information Retrieval, J. ACM, Vol. 8, p. 553, 1961.

54. Stiles, H. E.: Automatic Indexing and the Association Factor, Information Systems Compatibility. Newman, S. M., Ed., American University Technology of Management Series, Vol. 1, Ch. 13, p. 35, Spartan Books, 1965.

55. Taube, M.: A Note on the Pseudo-Mathematics of Relevance, American Documentation, Vol. 16, No. 2, p. 69, 1965.

56. Weinblatt, H. B.: Efficient Algorithms for Finding the Simple Cycles and Maximal Strongly Connected Regions of a Finite Directed Graph. Bell Labs. Tech. Memorandum MM 67-3343-7, 1967.

57. Winters, W. K.: A Modified Method of Latent Class Analysis for File Orgainzation in Information Retrieval, J. ACM, Vol. 12, No. 3, p. 356, 1965.

58. Wolfberg, M. S.: UP. L6 - An L^6 System for the IBM 7040. The Moore School of Electrical Engineering, University of Pennsylvania, Philadlephia, Pa., Internal publication, 1967.

59. Wolfberg, M. S.: Determination of Maximally Complete Subgraphs, Interim Technical Report, University of Pennsylvania, Contract NONR 555 (50) (Master's Thesis) 1965.

BOOKS

60. Harary, Norman, and Cartwright: Structure Models: An Introduction to the Theory of Directed Graphs. Wiley and Sons, New York, New York, 1965.

61. Ore, O.: Theory of Graphs, American Mathematical Society, Providence, Rhode Island, 1962.

62. Salton, Gerard: Automatic Information Organization and Retrieval. McGraw-Hill, New York, New York, 1968.

63. Vickery, B. C.: Classification and Indexing in Science, Second Edition, Butterworth and Co. (Publishers) Ltd., London, England.

APPENDIX A

USER'S MANUAL

University of Pennsylvania

THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

SOLER USER'S MANUAL

Philadelphia, Pennsylvania

August, 1971

INDEX OF CONTENTS

Introduction.....1

Processing of SOLER Commands.....1

Notes on this Manual.....2

Alphabetical Listing of SOLER Commands.....4

Illustrative Search Sequences

Appendix A.....40

SOLER USER'S MANUAL

INTRODUCTION

This manual serves as a user 's guide to the SOLER information retrieval system. The commands described here are the general user commands; no SOLER administrative commands are included.

PROCESSING OF SOLER COMMANDS

Commands are input to the SOLER system in a command stream. When the system is ready to accept a command stream, it will display "ENTER COMMAND"; the user can begin typing when a "*" is displayed. The command stream should consist of a series of one or more commands to be executed in sequence. The ENDCOMMAND symbol (see SET command) must separate commands; however, this symbol is not necessary after the last command in the stream. Transmission of command streams to the system may not exceed 80 characters in length; hence, any command stream which exceeds this limit must be broken into several transmissions. Any time a transmission ending with the CONTINUATION symbol (see SET command) is received by the system, it is stored and the user is allowed to continue his command stream with another transmission. Any error

condition encountered during processing of a command stream will result in the display of the appropriate message and termination of the command stream.

NOTES ON THIS MANUAL

1. Anything enclosed in parentheses in a command format is optional in that command.

2. <value> means any set of at most 200 characters.

3. <category name> is the single name of a category in any file of the data base. A category name may be ambiguous (i.e. used in different contexts within the data base). A category may be subcategorized; hence, a file name is a category name. An example of a file and its categories is shown in the description of the DESCRIBE command. ANYWHERE is a special category name used to denote the entire data base (only applicable where specified). The special category name CATEGORY indicates the attribute of being a category name (only applicable where specified). For example, RETRIEVE CATEGORY = NAME means "find the records in which NAME is a category".

4. <qualified category name> is a series of category names separated by periods. Each category must be a subcategory of the one to its left in the series. This type of expression allows for the unambiguous specification of a category name.

5. <qcn> is an abbreviation used in this manual to mean a category name which may or may not be qualified.

6. At all times, there is one list of records currently under consideration; this is called "the active list". After each retrieval command, the active list contains the list of records which satisfy the command. Data can be printed only from the records in the active list. The user can create the active list, save it, restore an old active list, or manipulate the pointer into the list by using the commands described in this manual.

FUNCTION OF COMMAND

Causes the system to process a RETRIEVE command based on the conditions specified in the retrieval expression; then the level-1 operator is applied to this list and to the current active list, thus producing a new active list. For example, if the "AND" operator is specified, the new active list consists of all records which appear on the current active list and also appear on the list produced by the current command.

FORMAT

APPLY (<level-1 operator>)<retrieval expression>

EXPLANATION OF FORMAT

1. <level-1 operator> is a logical operator. A list of these operators is available by issuing a SET LIST = OPER command.

2. If <level-1 operator> is omitted, the logical operator "AND" is assumed.

3. <retrieval expression> is the logical expression described as the argument of the RETRIEVE command.

APPLY command

EXAMPLE

```
 ENTER COMMAND
*RETRIEVE CATEGORY=PHONE DIRECTORY
 0000005 RECORDS HAVE BEEN RETRIEVED

 ENTER COMMAND
*APPLY AREA CODE = 215
 0000002 RECORDS HAVE BEEN RETRIEVED
 0000002 RECORDS RESULT FROM THIS APPLY

 ENTER COMMAND
*APPLY OR AREA CODE = 301
 0000001 RECORD HAS BEEN RETRIEVED
 0000003 RECORDS RESULT FROM THIS APPLY

 ENTER COMMAND
```

FUNCTION OF COMMAND

Displays to the user all values in the specified category in an alphabetical neighborhood of the specified value.

FORMAT

AROUND <value>(IN <qcn>)

AROUND <value>(,<qcn>)

EXPLANATION OF FORMAT

1. CATEGORY or ANYWHERE may be substituted for <qcn>. ANYWHERE is assumed if <qcn> is omitted.

EXAMPLE

```
 ENTER COMMAND
*AROUND JONES IN PHONE DIRECTORY.NAME
 JOAN
 JOHN
 JOHNSON
 JONES
 JONI
 JONSON
 JOSEPH

 ENTER COMMAND
```

BACKWARD command

FUNCTION OF COMMAND

Moves the active list pointer backward n entries in the list.

FORMAT

BACKWARD (n)

EXPLANATION OF FORMAT

1. n is any integer.
2. If n is omitted, 1 is assumed.

EXAMPLE

```
 ENTER COMMAND
*BACKWARD 3

 ENTER COMMAND
```

FUNCTION OF COMMAND

Displays to the user all values in the specified category which are alphabetically between the specified values.

FORMAT

BETWEEN <value> and <value>(IN <qcn>)

BETWEEN <value>,<value>(,<qcn>)

EXPLANATION OF FORMAT

1. CATEGORY or ANYWHERE may be substituted for <qcn>. ANYWHERE is assumed if <qcn> is omitted.

EXAMPLE

```
 ENTER COMMAND
*BETWEEN 200,400,AREA CODE
 212
 215
 301

 ENTER COMMAND
```

FUNCTION OF COMMAND

Instructs the system to print all data in the category that was specified in the most recent PRINT or LIST command (to the high-speed printer or the user's terminal, depending on which command was most recent), from the the next n records in the active list.

FORMAT

CONTINUE (n)

EXPLANATION OF FORMAT

1. n is any integer.

2. If n is omitted the number of entries currently remaining in the active list is assumed.

COMMENT command

FUNCTION OF COMMAND

Stores the specified comment for the SOLER administrator to review at a later time.

FORMAT

COMMENT <any comment>

EXPLANATION OF FORMAT

1. <any comment> is any set of remarks the user wishes to type in.

EXAMPLE

```
 ENTER COMMAND
*COMMENT THIS IS A MESSAGE TO THE ADMINISTRATOR

 ENTER COMMAND
```

CONTINUE command

EXAMPLE

```
 ENTER COMMAND
*RETRIEVE CATEGORY = DICTIONARY
 0000004 RECORDS HAVE BEEN RETRIEVED

 ENTER COMMAND
*PRINT 1, WORD


 RECORD NUMBER 000006

 DICTIONARY
 ..WORD ENTRY
 ....WORD
      NEW, A.

 ENTER COMMAND
*CONTINUE 2


 RECORD NUMBER 000015

 DICTIONARY
 ..WORD ENTRY
 ....WORD
      JERSEY, N.


 RECORD NUMBER 000016

 DICTIONARY
 ..WORD ENTRY
 ....WORD
      PHASE, N.

 ENTER COMMAND
```

FUNCTION OF COMMAND

Displays the structure of the specified categories; repeating categories are indicated by "(R)". If no argument is specified, a list of existing files is displayed.

FORMAT

DESCRIBE <qcn>(,<qcn>,...)

DESCRIBE

EXPLANATION OF FORMAT

1. Commas must separate <qcn>'s.

EXAMPLE

```
 ENTER COMMAND
*DESCRIBE
 EXISTING FILES ARE :
     DICTIONARY
     PHONE DIRECTORY
     POPULATION RECORDS

 ENTER COMMAND
*DESCRIBE PHONE DIRECTORY
 001 PHONE DIRECTORY
     002 NAME
         003 LAST NAME
         003 TITLE
         003 FIRST NAME
         003 MIDDLE NAME
     002 ADDRESS
         003 COMPANY
         003 STREET
         003 CITY
         003 STATE
         003 ZIP CODE
         003 COUNTRY
     002 PHONE NUMBER (R)
         003 AREA CODE
         003 NUMBER
         003 EXTENSION  (R)

 ENTER COMMAND
```

END command

FUNCTION OF COMMAND

Ends the session on the SOLER system. Control is passed back to the computer's operating system.

FORMAT

```
END
```

EXPLANATION OF FORMAT

1. Any arguments of this command will be ignored.

EXAMPLE

```
 ENTER COMMAND
*END
```

FUNCTION OF COMMAND

Releases the list with the specified identifier. Further reference to this list is no longer possible. However, the identifier can be used again in another SAVE command.

FORMAT

ERASE <identifier>

EXPLANATION OF FORMAT

1. <identifier> is any name used in a previous SAVE command.

EXAMPLE

```
 ENTER COMMAND
*ERASE MY POPULATION LIST
 FILE DESTROYED

 ENTER COMMAND
```

FUNCTION OF COMMAND

Moves the active list pointer forward n entries in the list.

FORMAT

FORWARD (n)

EXPLANATION OF FORMAT

1. n is any integer.
2. If n is omitted, 1 is assumed.

EXAMPLE

```
 ENTER COMMAND
*FORWARD 5

 ENTER COMMAND
```

FUNCTION OF COMMAND

Creates a new active list composed of the specified record numbers in numeric order.

FORMAT

GET <record number>(,<record number>,...)

EXPLANATION OF FORMAT

1. <record number> is any integer which represents an existing record.

2. Commas must separate <record number>'s.

EXAMPLE

```
 ENTER COMMAND
*GET 2,3,5,7

 ENTER COMMAND
*PRINT NONE

 RECORDS IN ACTIVE LIST

 000002
 000003
 000005
 000007

 END OF LIST ENCOUNTERED

 ENTER COMMAND
```

FUNCTION OF COMMAND

Interrupts the SOLER processing and enters "Interrupt mode". In this mode, the user is allowed to terminate a command stream by the RESTART command, or issue a SET command and resume processing by the RESUME command. The INTR command can only be issued after the user has depressed the "BREAK" or "ATTN" key on the keyboard and the operating system has responded with a "/".

FORMAT

INTR

EXPLANATION OF FORMAT

1. No arguments are allowed for this command.

EXAMPLE

```
 ENTER COMMAND
*GET 1,2,3

 ENTER COMMAND
*PRINT


 RECORD NUMBER 000001

 PHONE DIRECTORY
 ..NAME
 ....LAST NAME
      NEW
 ....FIRST NAME
      CHRIS
 ..ADDRESS
 ....STREET
      252 E 88 ST.
                        (break key depressed)
/INTR
 INTERRUPT MODE: ENTER SET, RESUME, OR RESTART
 COMMAND
*RESTART

 ENTER COMMAND
```

FUNCTION OF COMMAND

Initiates the same processing as a PRINT command, except that the output is directed to the high-speed printer instead of the user's terminal.

FORMAT

LIST (n,)NONE

LIST (n,)<qcn>(,<qcn>,...)

LIST (n)

EXPLANATION OF FORMAT

1. n is any integer.
2. If n is omitted the number of entries currently remaining in the active list is assumed.
3. Commas must separate <qcn>'s.

EXAMPLE

```
 ENTER COMMAND
*GET 2,5,17,18

 ENTER COMMAND
*FORWARD 2

 ENTER COMMAND
*LIST 1, WORD, NAME

 ENTER COMMAND
```

FUNCTION OF COMMAND

Instructs the system to print all data in the specified categories from the next n records in the active list. The output is directed to the user's terminal. A pointer to an entry in the active list is maintained; this pointer determines the place in the list at which to begin printing the n records. (Whenever a new active list is created, the pointer is set to the beginning of the list; printing from the list advances the pointer; the pointer can also be changed by using the FORWARD, BACKWARD, and RESET commands.) The NONE option causes printing of the active list itself. If no argument is specified, all categories of the remaining records in the active list are printed.

FORMAT

PRINT (n,)NONE

PRINT (n,)<qcn>(,<qcn>,...)

PRINT (n)

EXPLANATION OF FORMAT

1. n is any integer.
2. If n is omitted the number of entries currently remaining in the active list is assumed.
3. Commas must separate <qcn>'s.

EXAMPLE

```
 ENTER COMMAND
*GET 2,5,17,18

 ENTER COMMAND
*FORWARD 2

 ENTER COMMAND
*PRINT 1, WORD, DEFINITION ENTRY.DEFINITION


 RECORD NUMBER 000017

 DICTIONARY
 ..WORD ENTRY
 ....WORD
      FOREIGN
 ..DEFINITION ENTRY
 ....DEFINITION
      SITUATED OUTSIDE ONE'S OWN COUNTRY,
       PROVINCE, LOCALITY, ETC.
 ....DEFINITION
      COMING FROM OR HAVING TO DO WITH ANOTHER
       PERSON OR THING; NOT CHARACTERISTIC;
       AS, FORCE IS FOREIGN TO HIS NATURE
 ....DEFINITION
      EXCLUDED; NOT ADMITTED; HELD AT A
       DISTANCE

ENTER COMMAND
```

FUNCTION OF COMMAND

Produces an internal list of the categories specified in the command; all further retrieval and printing is limited to the listed categories and their subcategories. Since there is only one qualification list, each QUALIFY command destroys the old qualification list. If no argument is specified, the current qualification is removed.

FORMAT

QUALIFY <qcn>(,<qcn>,...)

QUALIFY

EXPLANATION OF FORMAT

1. Commas must separate <qcn>'s.

EXAMPLE

```
 ENTER COMMAND
*QUALIFY LAST NAME, TITLE

 ENTER COMMAND
*GET 8

 ENTER COMMAND
*PRINT


 RECORD NUMBER 000008

 PHONE DIRECTORY
 ..NAME
 ....LAST NAME
      KAIN
 ....TITLE
      REV.

 END OF LIST ENCOUNTERED

 ENTER COMMAND
*QUALIFY

 ENTER COMMAND
*PRINT NAME


 RECORD NUMBER 000008

 PHONE DIRECTORY
 ..NAME
 ....LAST NAME
      KAIN
 ....TITLE
      REV.
 ....FIRST NAME
      JAMES
 ....MIDDLE NAME
      L.

 END OF LIST ENCOUNTERED

 ENTER COMMAND
```

FUNCTION OF COMMAND

Instructs the system to reprocess the most recent APPLY or RETRIEVE command (even though other types of commands have been issued).

FORMAT

REPEAT

EXPLANATION OF FORMAT

1. Any arguments of this command will be ignored.

EXAMPLE

```
 ENTER COMMAND
*RESTRICT AREA CODE = 301 OR WORD = FOREIGN
 THE RESTRICTION LIST CONTAINS 0000002 RECORDS

 ENTER COMMAND
*RETRIEVE CATEGORY = AREA CODE
 0000005 RECORDS HAVE BEEN RETRIEVED
 0000001 RECORD RESULTS AFTER RESTRICTION

 ENTER COMMAND
*RESTRICT

 ENTER COMMAND
*REPEAT
 0000005 RECORDS HAVE BEEN RETRIEVED

 ENTER COMMAND
```

FUNCTION OF COMMAND

Sets the active list pointer to the beginning of the list.

FORMAT

RESET

EXPLANATION OF FORMAT

1. Any arguments of this command will be ignored.

EXAMPLE

```
 ENTER COMMAND
*RESET

 ENTER COMMAND
```

FUNCTION OF COMMAND

Instructs the system to terminate processing of the current command stream and accept a new command stream (applicable only in "interrupt mode").

FORMAT

RESTART

EXPLANATION OF FORMAT

1. No arguments are allowed for this command.

EXAMPLE

```
 ENTER COMMAND
*GET 1,2,3

 ENTER COMMAND
*PRINT


 RECORD NUMBER 000001

 PHONE DIRECTORY
 ..NAME
 ....LAST NAME
      NEW
 ....FIRST NAME
      CHRIS
 ..ADDRESS
 ....STREET
      252 E 88 ST.
                          (break key depressed)
/INTR
 INTERRUPT MODE: ENTER SET, RESUME, OR RESTART
 COMMAND
*RESTART

 ENTER COMMAND
```

FUNCTION OF COMMAND

Replaces the current active list with the list previously stored using the specified identifier.

FORMAT

RESTORE <identifier>

EXPLANATION OF FORMAT

1. <identifier> is any name used in a previous SAVE command.

EXAMPLE

```
 ENTER COMMAND
*RESTORE MY POPULATION LIST
 FILE ACTIVE

 ENTER COMMAND
*PRINT NONE

 RECORDS IN ACTIVE LIST

 000003
 000004
 000005
 000011
 000012
 000013
 000014

 END OF LIST ENCOUNTERED

 ENTER COMMAND
```

FUNCTION OF COMMAND

Causes the system to process a RETRIEVE command based on the conditions specified in the retrieval expression. The resulting active list is saved as the restriction list. Since there is only one restriction list, each RESTRICT command destroys the old restriction list. If the "*" argument is used, the current active list is saved as the restriction lis.. All further RETRIEVE or APPLY commands are limited to those records in the restriction list. If no argument is specified, the current restriction is removed.

FORMAT

RESTRICT <retrieval expression>

RESTRICT *

RESTRICT

EXPLANATION OF FORMAT

1. <retrieval expression> is the logical expression described as the argument of the RETRIEVE command.

EXAMPLE

```
 ENTER COMMAND
*RESTRICT AREA CODE = 215 OR WORD = FOREIGN
 THE RESTRICTION LIST CONTAINS 0000003 RECORDS

 ENTER COMMAND
*RETRIEVE CATEGORY = AREA CODE
 0000005 RECORDS HAVE BEEN RETRIEVED
 0000002 RECORDS RESULT AFTER RESTRICTION

 ENTER COMMAND
*RESTRICT

 ENTER COMMAND
*REPEAT
 0000005 RECORDS HAVE BEEN RETRIEVED

 ENTER COMMAND
*RESTRICT *

 ENTER COMMAND
*REPEAT
 0000005 RECORDS HAVE BEEN RETRIEVED
 0000005 RECORDS RESULT AFTER RESTRICTION

 ENTER COMMAND
```

FUNCTION OF COMMAND

Instructs the system to resume processing of the current command stream (applicable only in "Interrupt mode"). This command is particularly useful to resume processing after a SET command has been issued in "Interrupt mode".

FORMAT

RESUME

EXPLANATION OF FORMAT

1. No arguments are allowed for this command.

EXAMPLE

```
 ENTER COMMAND
*SET TRACES=ON

 ENTER COMMAND
*RETRIEVE ANYWHERE=NEW OR FOREIGN
       N=V OR V
         *  *
 000001  *  =
 000000  *  =
 000001  =  *
 000000  *  =
 000001  =  *
 000001  *  =
                          (break key depressed)
/INTR
 INTERRUPT MODE: ENTER SET, RESUME, OR RESTART
 COMMAND
*SET TRACES=OFF
 INTERRUPT MODE: ENTER SET, RESUME, OR RESTART
 COMMAND
*RESUME
 0000012 RECORDS HAVE BEEN RETRIEVED

 ENTER COMMAND
```

FUNCTION OF COMMAND

Produces a list (called the active list) of all the records which contain data that satisfies the specified conditions. There is only one active list; hence, every RETRIEVE command creates a list which replaces the current active list. The set of retrievable records may be limited by the RESTRICT command. The scope of the logical expression may be limited by the QUALIFY command.

FORMAT

RETRIEVE <qcn>=<expression of values>(<level-1 operator><qcn>=<expression of values>...)

EXPLANATION OF FORMAT

1. <expression of values> is defined as : <value>(<level-2 operator><value>...).

2. A space between <value>'s in a phrase is considered a <level-2 operator>.

3. <level-1 operator> and <level-2 operator> are logical operators. A list of these operators is available by issuing a SET LIST = OPER command.

4. Parentheses are allowed to specify the order of operations.

5. <value> should be enclosed in single quotes if it contains non-alphanumeric characters (e.g. hyphen), or if it is a reserved word (e.g. the name of an operator or function).

6. <value> may be replaced by : <function>(<value>(,<value>,...)).

7. <function> is an alphabetic browsing function (e.g. AROUND, BETWEEN, TRUNCATE). The arguments of a function must be enclosed in parentheses and separated by commas.

8. CATEGORY or ANYWHERE may be substituted for <qcn>.

EXAMPLE

```
ENTER COMMAND
*RETRIEVE IDIOM = POLICY OF A COUNTRY
0000001 RECORD HAS BEEN RETRIEVED

ENTER COMMAND
*PRINT IDIOM


RECORD NUMBER 000017

DICTIONARY
..IDIOM
....PHRASE
      FOREIGN AFFAIRS
....DEFINITION
      MATTERS CONCERNING POLICY OF A COUNTRY IN
       ITS RELATIONS WITH OTHER COUNTRIES
....PHRASE
      FOREIGN OFFICE
....DEFINITION
      THE DEPARTMENT OF GOVERNMENT IN CHARGE OF
       FOREIGN AFFAIRS

END OF LIST ENCOUNTERED

ENTER COMMAND
*RETRIEVE NAME=NEWS(STATE=NEW YORK-CITY=ALBANY)
0000001 RECORD HAS BEEN RETRIEVED

ENTER COMMAND
*APPLY STREET = 42ND ST
NO RECORDS SATISFY THIS RETRIEVE
0000000 RECORDS RESULT FROM THIS APPLY

ENTER COMMAND
*RETRIEVE WORD = TRUNCATE(ELECTRO) OR 'ON-LINE'
0000002 RECORDS HAVE BEEN RETRIEVED

ENTER COMMAND
```

FUNCTION OF COMMAND

Stores the current active list internally, using the specified identifier. In order to reference it later, a RESTORE command must be issued.

FORMAT

SAVE <identifier>

EXPLANATION OF FORMAT

1. <identifier> is any name which the user assigns.

EXAMPLE

```
 ENTER COMMAND
*RETRIEVE CATEGORY = POPULATION RECORDS
 0000007 RECORDS HAVE BEEN RETRIEVED

 ENTER COMMAND
*PRINT NONE

 RECORDS IN ACTIVE LIST

 000003
 000004
 000005
 000011
 000012
 000013
 000014

 END OF LIST ENCOUNTERED

 ENTER COMMAND
*SAVE MY POPULATION LIST
 FILE HAS BEEN SAVED

 ENTER COMMAND
```

FUNCTION OF COMMAND

Changes the settings of the specified user-controlled conditions. The LIST option causes printing of the specified keywords, their abbreviations, possible values, and current values. If no keywords are specified for the LIST option, all keywords, abbreviations, and values are printed.

FORMAT

SET <keyword>=<value>(,<keyword>=<value>,...)

SET LIST(=<keyword>,<keyword>,...)

EXPLANATION OF FORMAT

1. Commas must separate <keyword>'s or <keyword> = <value> pairs.

2. <keyword>'s are described in the accompanying list; <value>'s are described by issuing a SET LIST command.

3. Each <keyword> has an equivalent abbreviation.

KEYWORD	DESCRIPTION OF VALUE
CONTINUATION	a single symbol used to denote continuation of any command stream from one transmission to the next; cannot have the same value as ENDCOMMAND.
ENDCOMMAND	a single symbol used to separate commands in a command stream; cannot have the same value as CONTINUATION.
FIELDNAMES	one of four settings which specify the extent to which category names are printed along with data; the settings allow printing of all or none of the category names associated with a data item, all subcategories of the category being printed, or only the name of the category which contains the data itself.
FUNCLIMIT	an integer which sets a limit on the amount of searching done to evaluate any function.
INDENTFACTOR	an integer used for indenting category names and data when printing; the value must be less than LINELENGTH.
LINELENGTH	an integer which determines the maximum line length for all printing.
OPERATOR	a set of five values which defines a new logical operator for the current session; the values consist of the name and precedence of the new operator, the name of the subroutine to be used for performing the operation, and indications that the operator is either level-1 or level-2 or both.
PRECEDENCE	an indication of a new precedence for an existing operator for the current session.

KEYWORD	DESCRIPTION OF VALUE
PRINT-DATA	one of three settings which allow the SOLER administrator to display data being updated on the terminal, high-speed printer, or not at all; applicable during file update only.
RECORDNUMBER	a yes-no setting which specifies whether or not record numbers are to be displayed with data being printed.
SKIP-TO-TOP	a yes-no setting which specifies whether or not each record (being printed on the high-speed printer) starts at the top of a new page.
SYSDTA->FILE	one of two settings which specify whether or not the user's commands (from SYSDTA) are echoed to a cataloged file.
SYSDTA->LST	one of two settings which specify whether or not the user's commands (from SYSDTA) are echoed to the high-speed printer (SYSLST).
SYSDTA->OUT	one of two settings which specify whether or not the user's commands (from SYSDTA) are echoed to the user's terminal (SYSOUT).
SYSLST->FILE	one of three settings which specify whether or not the data being printed on the high-speed printer (SYSLST) is echoed or switched to a cataloged file.
SYSOUT->FILE	one of three settings which specify whether or not the data being printed on the user's terminal (SYSOUT) is echoed or switched to a cataloged file.
SYSOUT->LST	one of three settings which specify whether or not the data being printed on the user's terminal (SYSOUT) is echoed or switched to the high-speed printer (SYSLST).

KEYWORD	DESCRIPTION OF VALUE
TRACELEVEL1	a yes-no setting which specifies whether or not the result of each level-1 operation is traced on the user's terminal.
TRACELEVEL2	a yes-no setting which specifies whether or not the result of each level-2 operation is traced on the user's terminal.
TRACELEVELG	a yes-no setting which specifies whether or not the result of each operation generated by subcategories or functions is traced on the user's terminal.
TRACES	a yes-no setting which specifies whether or not the results of all operations are traced on the user's terminal; in essence, TRACELEVEL1, TRACELEVEL2, and TRACELEVELG are all set to yes or no.
TRACELIMIT	an integer which sets a limit on the trace of each operator; when the number of records resulting from any operation is greater than this integer, the trace is not displayed.
TRACESYMBOL	a single symbol used in displaying the trace; cannot be "=".

FUNCTION OF COMMAND

Displays to the user all values in the specified category for which the specified value forms the word trunk (i.e. the first string of characters).

FORMAT

TRUNCATE <value>(IN <qcn>)

TRUNCATE <value>(,<qcn>)

EXPLANATION OF FORMAT

1. CATEGORY or ANYWHERE may be substituted for <qcn>. ANYWHERE is assumed if <qcn> is omitted.

EXAMPLE

```
 ENTER COMMAND
*TRUNCATE ELECTRO
 ELECTROCARDIOGRAM
 ELECTROCUTE
 ELECTROENCEPHALOGRAM
 ELECTROLYSIS
 ELECTROMAGNETIC
 ELECTRON
 ELECTRONIC

 ENTER COMMAND
*TRUNCATE ELECTRO IN DIAGNOSIS
 ELECTROCARDIOGRAM
 ELECTROENCEPHALOGRAM

 ENTER COMMAND
*TRUNCATE DESCR,CATEGORY
 DESCRIPTION
 DESCRIPTIVE

 ENTER COMMAND
```

FUNCTION OF COMMAND

Displays to the user the names of all fields (in the data base) in which the indicated value appears as data, and the number of records in which the indicated value is the name of a category. The data base may be limited by the QUALIFY command.

FORMAT

WHERE <value>(,<value>,...)

EXPLANATION OF FORMAT

1. <value> cannot include commas.
2. Commas must separate <value>'s.

EXAMPLE

```
 ENTER COMMAND
*QUALIFY PHONE DIRECTORY, DICTIONARY

 ENTER COMMAND
*WHERE 215,NAME,NEW

 ********
 215
 ********

 OCCURS IN THE DATA BASE :
 PHONE DIRECTORY
 WITHIN THESE FIELDS ( # RECORDS)

 AREA CODE  (000002)

 ********
 NAME
 ********
 IS A CATEGORY CONTAINING DATA IN 000012 RECORDS


 ********
 NEW
 ********

 OCCURS IN THE DATA BASE :
 DICTIONARY
 WITHIN THESE FIELDS ( # RECORDS)

 WORD (000001)
 DEFINITION (000001)
 PHRASE (000001)

 OCCURS IN THE DATA BASE :
 PHONE DIRECTORY
 WITHIN THESE FIELDS ( # RECORDS)

 LAST NAME  (000001)
 COMPANY  (000001)
 CITY (000002)
 STATE  (000001)

 ALSO OCCURS OUTSIDE CURRENT QUALIFICATION

 ENTER COMMAND
```

APPENDIX A

Illustrative Search Sequences

```
1. /LOGON user-id,acct#
   %C 0223 LOGON ACCEPTED FROM LINE #nnn AT time
   ON date, TSN nnnn ASSIGNED.
2. /DO RETRIEVE
3. %P500 LOADING

4.  ENTER COMMAND
5. *DESCRIBE
    EXISTING FILES ARE :
        DICTIONARY
        PHONE DIRECTORY
        POPULATION RECORDS

    ENTER COMMAND
6. *DESCRIBE PHONE DIRECTORY
    001 PHONE DIRECTORY
        002 NAME
            003 LAST NAME
            003 TITLE
            003 FIRST NAME
            003 MIDDLE NAME
        002 ADDRESS
            003 COMPANY
            003 STREET
            003 CITY
            003 STATE
            003 ZIP CODE
            003 COUNTRY
        002 PHONE NUMBER (R)
            003 AREA CODE
            003 NUMBER
            003 EXTENSION  (R)

   ENTER COMMAND
7. *QUALIFY PHONE DIRECTORY
```

---

1. After dialing into the computer, the user must identify himself to the operating system. The system responds when the logon is accepted.
2. The user initiates SOLER retrieval with the DO command to the operating system.
3. The operating system informs the user that SOLER is being loaded.
4. SOLER is ready to accept a command from the user.
5. The user asks for a list of the files in this data base.
6. The user asks for a description of one of the files in the data base. This description is displayed in the level structure of the file definition.
7. The user instructs SOLER to limit his retrieval to the PHONE DIRECTORY file in the data base.

```
    ENTER COMMAND
8. *RETRIEVE NAME = CHRIS
    0000001 RECORD HAS BEEN RETRIEVED

    ENTER COMMAND
9. *PRINT

    RECORD NUMBER 000001

    PHONE DIRECTORY
    ..NAME
    ....LAST NAME
         NEW
    ....FIRST NAME
         CHRIS
    ..ADDRESS
    ....STREET
         252 E 88 ST.
    ....CITY
         NEW YORK
    ....STATE
         NEW YORK
    ....ZIP CODE
         10017
    ..PHONE NUMBER
    ....AREA CODE
         212
    ....NUMBER
         238-3145

    END OF LIST ENCOUNTERED
```

---

8. The retrieve command initiates retrieval. The NAME = CHRIS clause specifies that all records should be retrieved which have the value CHRIS in the NAME category. The system tells the user how many records result.
9. The user tells the system to print this record. SOLER prints the internal record number of the record retrieved. Then, the names of the data categories are printed; The indentation indicates which categories are subordinate to other categories in the tree-structured record. The whole data record is printed.

```
     ENTER COMMAND
10. *BETWEEN A AND H IN NAME
     A
     ALBERT
     ALPHONSE
     CHRIS
     DERRICK
     FOREIGN
     GEORGE

     ENTER COMMAND
11. *RETR NAME=BETWEEN(A,H) AND AREA CODE = 301
     0000001 RECORD HAW BEEN RETRIEVED

     ENTER COMMAND
12. *SET FIELDNAMES=IMMED

     ENTER COMMAND
13. *PRINT NAME,ADDRESS

     RECORD NUMBER 000010

     ....LAST NAME
          SMITH
     ....FIRST NAME
          GEORGE
     ....MIDDLE NAME
          HAROLD
     ....STREET
          200 N MAIN ST.
     ....CITY
          CATONSVILLE
     ....STATE
          MARYLAND
     ....ZIP CODE
          21229

     END OF LIST ENCOUNTERED
```

10. The user asks to see all values in the NAME category between the letters A and H.
11. Now the user wants to retrieve all records with a NAME value between A and H which also have an AREA CODE value of 301. SOLER tells the user the number of records retrieved.
12. The user specifies that only immediate category names are to be printed in subsequent PRINT's.
13. Request is issued for printing of the NAME and ADDRESS categories from the record retrieved.

```
     ENTER COMMAND
14. *RETR CATEGORY=PHONE DIRECTORY
     0000005 RECORDS HAVE BEEN RETRIEVED

     ENTER COMMAND
15. *LIST

     ENTER COMMAND
16. *QUALIFY

     ENTER COMMAND
17. *RETR CATEGORY=DICTIONARY
     0000004 RECORDS HAVE BEEN RETRIEVED

     ENTER COMMAND
18. *PRINT NONE

     RECORDS IN ACTIVE LIST

     000006
     000015
     000016
     000017

     END OF LIST ENCOUNTERED
```

---

14. The user wishes to find all records which contain data for the PHONE DIRECTORY category; that is, all records in the PHONE DIRECTORY file are desired.
15. The user specifies that all five records retrieved should be printed on the high-speed printer. No printing is directed to the terminal.
16. The user instructs SOLER to remove the current qualification; that is, the user wishes to retrieve from the entire data base again.
17. The user desires all records in the DICTIONARY file.
18. The internal record numbers of the currently active records are displayed.

```
     ENTER COMMAND
19. *SET RECORDNUMBER=N,FN=NONE

     ENTER COMMAND
20. *PRINT WORD
          NEW, A.
          JERSEY, N.
          PHASE, N.
          FOREIGN

     END OF LIST ENCOUNTERED

     ENTER COMMAND
21. *SET FN=1

     ENTER COMMAND
22. *RESET;FORWARD 2;PRINT 1,WORD ENTRY,SYNONYMS
     ....WORD
          PHASE, N.
     ....ORIGIN
          MOD. L. PHASIS; GR. PHASIS, FROM PHAINESTHAI,
           TO APPEAR
```

---

19. The user requests that record numbers and all category names be suppressed in firther printing.
20. The WORD category from each active record is printed without category names or record numbers.
21. The display of terminal category names is restored for further printing.
22. The user issues three commands; the first sets the pointer to the first active record; the second advances the pointer to the third active record; and the third requests printing of WORD ENTRY and SYNONYMS from the next active record.

```
     ENTER COMMAND
23. *RETRIEVE CATEGORY = PHONE DIRECTORY - ZIP CODE OR =
     *ADDRESS = PHILADELPHIA OR NEW YORK
      0000003 RECORDS HAVE BEEN RETRIEVED

     ENTER COMMAND
24. *PRINT LAST NAME
     ....LAST NAME
          NEW
     ....LAST NAME
          ZIGERT
     ....LAST NAME
          DERRICK

     END OF LIST ENCOUNTERED

     ENTER COMMAND
25. *APPLY ADDRESS=PHILADELPHIA
     0000001 RECORD HAS BEEN RETRIEVED
     0000001 RECORD RESULTS FROM THIS APPLY

     ENTER COMMAND
26. *PRINT NAME,ADDRESS
     ....LAST NAME
          ZIGERT
     ....TITLE
          JR.
     ....FIRST NAME
          ALBERT
     ....MIDDLE NAME
          FOREIGN
     ....COMPANY
          UNIV. OF PA.
     ....STREET
          200 SOUTH 33 ST.
     ....CITY
          PHILADELPHIA
     ....STATE
          PENNSYLVANIA
     ....ZIP CODE
          19104

     END OF LIST ENCOUNTERED
```

---

23. A two-line retrieval request is issued; the continuation character is added at the end of the first line.
24. The LAST NAME entries in the active records are printed.
25. The additional constraint "ADDRESS=PHILADELPHIA" is imposed on the active list.
26. Data from the resulting record is printed.

```
     ENTER COMMAND
27. *RESTRICT CATEGORY=DICTIONARY
     THE RESTRICTION LIST CONTAINS 0000004 RECORDS

     ENTER COMMAND
28. *RETRIEVE ANYWHERE=FOREIGN
     0000002 RECORDS HAVE BEEN RETRIEVED
     0000001 RECORD RESULTS AFTER RESTRICTION

     ENTER COMMAND
29. *PRINT IDIOM
     ....PHRASE
          FOREIGN AFFAIRS
     ....DEFINITION
          MATTERS CONCERNING POLICY OF A COUNTRY IN ITS
           RELATIONS WITH   OTHER COUNTRIES
     ....PHRASE
          FOREIGN OFFICE
     ....DEFINITION
          THE DEPARTMENT OF GOVERNMENT IN CHARGE OF
           FOREIGN AFFAIRS

     END OF LIST ENCOUNTERED

     ENTER COMMAND
30. *RETRIEVE WORD=TRUNCATE(ELECTRO)
     NO RECORDS SATISFY THIS RETRIEVE

     ENTER COMMAND
31. *END
32. /LOGOFF
```

27. All records in the DICTIONARY file are retrieved and the resulting list is used to restrict further retrieval.
28. The user requests all records in the entire data base in which the value FOREIGN occurs. SOLER notes that two such records are found in the data base, but only one is in the restriction list of records.
29. The user asks to see the IDIOM category in the resulting record.
30. The user requests all records that have values in the WORD category which begin with the letters ELECTRO. SOLER informs him that no such records exist.
31. Finished with his work, the user ends the SOLER session.
32. The LOGOFF command to the operating system disconnects the user.